



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΕΧΝΟΛΟΓΙΑ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ**

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Dynamic Scaling of Parallel Stream Joins on the Cloud

Εμμανουήλ Ιωάννη Αγγελογιαννόπουλος

Επιβλέπων: Αλέξης Δελής, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

Μάιος 2017

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Dynamic Scaling of Parallel Stream Joins on the Cloud

Εμμανουήλ Ι. Αγγελογιαννόπουλος

A.M.: M1368

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής**, Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Μέμα Ρουσσοπούλου**, Αναπληρώτρια Καθηγήτρια

Μάιος 2017

ΠΕΡΙΛΗΨΗ

Οι μεγάλοι όγκοι δεδομένων που παράγονται από πολλές αναδυόμενες εφαρμογές και συστήματα απαιτούν την πολύπλοκη επεξεργασία ροών δεδομένων υψηλής ταχύτητας σε πραγματικό χρόνο. Η σύζευξη δεδομένων ροών είναι η αντίστοιχη διαδικασία σύζευξης των συμβατικών βάσεων δεδομένων και συγκρίνει τις πλειάδες που προέρχονται από διαφορετικές σχεσιακές ροές. Ο συγκεκριμένος operator χαρακτηρίζεται ως υπολογιστικά ακριβός και ταυτόχρονα εξαιρετικά σημαντικός για την ανάλυση δεδομένων σε πραγματικό χρόνο. Η αποτελεσματική και κλιμακούμενη επεξεργασία των συζεύξεων δεδομένων ροών μπορεί να γίνει εφικτή από τη διαθεσιμότητα ενός μεγάλου αριθμού κόμβων επεξεργασίας σε ένα παράλληλο και καταναμημένο περιβάλλον. Επιπλέον, τα υπολογιστικά νέφη έχουν εξελιχθεί ως μια ελκυστική πλατφόρμα για την επεξεργασία δεδομένων μεγάλης κλίμακας, κυρίως λόγω της έννοιας της ελαστικότητας. Με τα υπολογιστικά νέφη δίνεται η δυνατότητα εκμίσθωσης εικονικής υπολογιστικής υποδομής, η οποία μπορεί να χρησιμοποιηθεί για όσο χρόνο χρειάζεται με δυναμικό τρόπο. Στη συγκεκριμένη εργασία υιοθετούμε τις βασικές ιδέες και τα χαρακτηριστικά των Qian Lin et al. από το έργο τους "Scalable Distributed Stream Join Processing". Η βασική ιδέα που παρουσιάζεται σε αυτό το έργο είναι το μοντέλο join-biclique το οποίο οργανώνει τις μονάδες επεξεργασίας ενός υπολογιστικού cluster ως έναν ολοκληρωμένο διμερές γράφο. Με βάση αυτή την ιδέα, αναπτύξαμε και υλοποιήσαμε ένα σύνολο αλγορίθμων που σχεδιάστηκαν ως microservices σε περιβάλλον software containers. Οι αλγόριθμοι εκτελούν την επεξεργασία και σύζευξη ροών δεδομένων και μπορούν να κλιμακωθούν οριζόντια. Πραγματοποιήσαμε τα πειράματά μας σε περιβάλλον υπολογιστικού νέφους στο Google Container Engine χρησιμοποιώντας πλατφόρμα Kubernetes και Docker containers.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Καταναμημένη Επεξεργασία Ροών Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: επεξεργασία ροών δεδομένων, καταναμημένα συστήματα, υπολογιστικό νέφος

ABSTRACT

The large and varying volumes of data generated by many emerging applications and systems demand the sophisticated processing of high speed data streams in a real-time fashion. Stream joins is the streaming counterpart of conventional database joins and compares tuples coming from different streaming relations. This operator is characterized as computationally expensive and also quite important for real-time analytics. Efficient and scalable processing of stream joins may be enabled by the availability of a large number of processing nodes in a parallel and distributed environment. Furthermore, clouds have evolved as an appealing platform for large-scale data processing mainly due to the concept of elasticity; virtual computing infrastructure can be leased on demand and used for as much time as needed in a dynamic manner. For this thesis project, we adopt the main ideas and features of Qian Lin et al. in their paper “Scalable Distributed Stream Join Processing”. The basic idea presented in that paper is the join-biclique model which organizes the processing units of a cluster as a complete bipartite graph. Based on that idea, we developed and carried out a set of algorithms designed as containerized microservices, which perform stream join processing and can be scaled horizontally on demand. We performed our experiments on Google Container Engine using Kubernetes orchestration platform and Docker containers.

SUBJECT AREA: Distributed Stream Join Processing

KEYWORDS: online stream join processing, distributed systems, cloud computing, containers

Στην οικογένεια μου.

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστώ τον καθηγητή κ. Αλέξη Δελή για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο θέμα. Επίσης τον ευχαριστώ για την καθοδήγηση και τη γενικότερη υποστήριξη καθ'όλη τη διάρκεια των σπουδών μου. Επιπλέον, θα ήθελα να ευχαριστήσω τους συναδέλφους κ. Νίκο Φούντα και κ. Δημήτρη Παπαδημητρίου για τις εποικοδομητικές συζητήσεις σε διάφορα τεχνικής φύσεως θέματα κατά τη διάρκεια της εργασίας.

Επίσης, ευχαριστώ όλους τους φίλους και συναδέλφους για την υποστήριξή τους. Ιδιαίτερες ευχαριστίες στη Δήμητρα για την ενθάρρυνση, υποστήριξη και ανοχή της σε όλη τη διάρκεια της εργασίας. Τέλος, η συγκεκριμένη εργασία δεν θα ήταν εφικτή χωρίς την έμπρακτη βοήθεια και υποστήριξη της οικογένειάς μου.

TABLE OF CONTENTS

FOREWORD	11
1. INTRODUCTION	12
1.1 The emergence of streaming.....	12
1.2 Stream Joins	12
1.3 Cloud computing	12
1.4 Goals of the project	13
1.5 Structure of thesis	13
2. BACKGROUND	15
2.1 Basic Concepts on Data Streams	15
2.2 Online Joins over Data Streams.....	16
2.3 Related Work on Parallel Stream Joins	17
2.4 The join-biclique model.....	18
2.4.1 Comparison with Join-Matrix Model	20
3. ELASTIC-BICLIQUE SYSTEM ARCHITECTURE	22
3.1 System Design	22
3.1.1 Router.....	23
3.1.2 Joiner.....	23
3.1.3 RabbitMQ Broker	25
3.2 Dataflow Control	28
3.3 Tuple Ordering Protocol	29
4. SYSTEM IMPLEMENTATION	32
4.1 Spring Boot	32
4.2 Spring Cloud Stream	32
4.2.1 Main Concepts	33

4.3	Implementation Analysis	35
4.4	Docker containers.....	37
4.5	Kubernetes	38
4.6	Google Container Engine.....	39
5.	DEPLOYMENT AND EXPERIMENTS	40
5.1	Setup and Deployment.....	40
5.2	Experiments	43
6.	CONCLUSION	49
6.1	Future Work.....	49
	ABBREVIATIONS - ACRONYMS	50
	REFERENCES	51

LIST OF FIGURES

Figure 1: Windowed Join [10]	16
Figure 2: Complete Bipartite Graph	18
Figure 3: Stream Join Models [3].....	20
Figure 4: Overall architecture of elastic-biclique	22
Figure 5: Chained in-memory index [3].....	24
Figure 6: Message Flow in AMQ model.....	26
Figure 7: AMQP Layers [45]	27
Figure 8: Arrival order of tuples r and s. [3].....	29
Figure 9: Spring Cloud Stream abstractions [19]	33
Figure 10: Concept of Consumer Group [19].....	34
Figure 11: Partitioning Concept [19]	35
Figure 12: RabbitMQ binder [19]	35
Figure 13: VM vs Containers	38
Figure 14: Cluster information	40
Figure 15: Cluster bootstrap	41
Figure 16: Kubernetes Services	41
Figure 17: Kubernetes Deployments	42
Figure 18: RabbitMQ idle queues	43
Figure 19: Horizontal Pod Autoscaler	43
Figure 20: Dynamic Scaling based on CPU utilization.....	45
Figure 21: Dynamic Scaling based on Memory Load	47

LIST OF TABLES

Table 1: Join-biclique symbols.....	19
-------------------------------------	----

FOREWORD

Submitted in part fulfillment of the requirements for the Master's degree in Computer System's Technologies at the National and Kapodistrian University of Athens, May 2017.

1. INTRODUCTION

1.1 The emergence of streaming

Today's information processing systems face formidable challenges as they are presented with new data at ever increasing rates. The widespread adoption of the Internet and the world-wide emergence of large cyber-physical systems and applications demands for near real-time processing of continuous data streams [1]. A broad range of applications and other sources may produce data streams, such as smart grids, enhanced medical systems, telemetry from Internet of Things (IoT) devices, clickstreams, stock trading and fraud detection algorithms etc. It is increasingly important to process and provide efficient real-time analytics for such applications and systems. In this context, the streaming paradigm introduces new semantics and also raises new operational challenges [2].

1.2 Stream Joins

In the streaming computing paradigm, graphs of stream operators are employed to process the incoming data in an online fashion. The stream joins are among the most important and expensive operators [6]. Compared to one-time joins in traditional DBMS's, continuous stream joins differ substantially in their semantics. They perform comparisons between tuples coming from different logical data streams rather than database relations. Since the size of the stream is potentially unbounded, the state of the data is not known in advance, so responses depend on the set of stream tuples available during join evaluation. Normally, streaming tuples are retained in main memory and not stored on persistent disk and thus it is not feasible to remember the full history of the rapidly accumulating stream elements due to resource limitations. To this end, the most common approach to perform joins in the streaming context is to introduce *windows* of data. Such constructs focus on the latest arriving data by exploiting a sense of ordering between them, usually established by a unique timestamp for each element [10].

Several parallelization techniques of stream joins have been proposed in the literature, both shared memory and shared nothing approaches. Shared memory techniques allow for parallel stream joins to scale-up within individual nodes, while shared nothing techniques allow for scaling-out parallel stream joins in a multi-node cluster. As emphasized by Gibbons in [8], scaling both out and up is crucial to efficiently address the challenges in the Big Data context and improve performance by orders of magnitude. In this thesis project, we seek to scale-out parallel stream joins into a scalable cluster. The goal is to exploit the elasticity (auto-scaling) of a cloud environment in order to deal with varying rates of the input streams. We adopt the join-biclique stream join model as presented in [3].

1.3 Cloud computing

Cloud computing has been one of the most hyped trends of the last few years. Initially introduced by Amazon [12], now cloud services are offered by numerous providers [13] [14]. Cloud computing is a broad term that encompasses many different aspects of a modern paradigm for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services). These resources can be rapidly provisioned and released with minimal management effort and provider interaction. The cloud computing model promotes the availability, rapid elasticity and is composed of three service models: SaaS (software as a service), PaaS (platform as a service) and IaaS (infrastructure as a service) and four

deployment models: private cloud, public cloud, community cloud and hybrid cloud [15]. IaaS describes a service that provides access to computing resources in a virtualized environment (e.g., computation, storage, and network) on demand. The administration of the system lies mostly with the user. PaaS takes some of the administration away from the user and allows some (limited) programming of the resources. An example for this is Google’s App Engine. Finally, and probably most exposed to the general public are SaaS applications. These are offerings such as Slack and Microsoft Office 365 applications. They offer little to no customization but the convenience of storing data off-site. We are interested in applying IaaS services to the computation of stream joins.

1.4 Goals of the project

In this thesis project we seek to address the problem of the distributed stream join processing in a cloud environment. In particular, we adopt the main ideas presented in the paper “Scalable Distributed Stream Join Processing” by Qian Lin et al. appeared on the 2015 ACM SIGMOD International Conference on Management of Data [3]. We are interested in providing an alternative implementation of the ideas presented using:

- Cutting-edge technologies and design principles in software engineering, such as event-driven micro-services and software containers.
- An elastic infrastructure comprising the processing units of the stream join engine, deployed on an IaaS cloud provider.

These are the basic goals of our project. We want to create a multi-node cluster with elastic characteristics, which is able to scale in and out on demand, depending on the stream workload traffic. The auto-scaling decisions should be set by the operator of the cloud application depending on several performance criteria of the processing units (e.g. CPU utilization, requests per second etc.).

The main ideas presented in the aforementioned paper include the join-biclique model, which organizes the stream join processing units of the cluster as a complete bipartite graph or biclique. The authors claim that this model is scalable and elastic with respect to the network size and efficient in terms of resource requirements. Their original attempt to implement a join engine based on that model is termed BiStream [16]. BiStream is based on Apache Storm; a distributed real-time computation framework [17]. Unfortunately, the current version of Storm does not support auto-scaling of processing units inside a topology. Thus, we opted for an implementation of the shared-nothing stream join model on an IaaS cloud provider, which natively offers dynamic scaling as a service. In order to achieve our goals, we developed a set of algorithms based on the original ideas of join-biclique and used state-of-the-art tools to achieve our goals. Such tools include Spring Boot [18], Spring Cloud Stream [19], Docker containers [20] and Kubernetes [21]. We then deployed the algorithms on Google Cloud platform; an IaaS cloud offered by Google Inc.

1.5 Structure of thesis

The rest of the thesis is structured as follows: Chapter 2 provides some background on the problem of streaming joins and discusses related work. Furthermore, it introduces the idea and model of join-biclique. It also presents a basic comparison with a similar architecture for shared-nothing stream joins. Chapter 3 presents the chosen architecture to implement this stream join model and describes our design

considerations. Chapter 4 describes the implementation of our design and the technology tools that we used. Chapter 5 presents our deployment, experiments and results and Chapter 6 concludes the thesis.

2. BACKGROUND

This chapter provides background information on the technologies and concepts relevant to this thesis. Section 2.1 introduces important definitions and basic concepts on data streams. Section 2.2 describes the online stream join operator in detail. Section 2.3 provides a relevant literature review on parallel stream joins. Section 2.4 introduces the join-biclique model for joining data streams in a distributed environment and provides a basic comparison with the join-matrix model.

2.1 Basic Concepts on Data Streams

We are following the description and semantics of data streams commonly referred in related literature [27] [28] [29]. Items of a data stream are often represented as relational tuples. Patroumpas et al. in [10] provide the following definitions relevant to data streams:

Definition 1 (Tuple Schema): *A tuple schema E of streaming items is represented as a set of finite elements $\langle e_1, e_2, \dots, e_N \rangle$. Each element e_i is termed attribute and its values may originate from a specific data type. Every tuple is an instance of the schema and is characterized by the values of its attributes.*

Normally, a timestamp value may be attached to every streaming tuple to the corresponding attribute as a way of determining a natural ordering between the items which flow into a stream processing system. Other ways of defining the order among tuples may be specified, e.g. a sequence number attribute. Both flavors of ordering may be covered from the following definition:

Definition 2 (Time Domain T): *A time domain T is defined as an infinite set of discrete ordered time constants $t \in T$. A time interval $[t_1, t_2] \in T$ may be specified as a set of all distinct time instants $t \in T$ for which the following comparison holds: $t_1 \leq t \leq t_2$.*

In similar spirit, we can now define the concept of a data stream:

Definition 3 (Data Stream): *A data stream may be defined as a mapping $S : T \rightarrow 2^R$, where at each instance $t \in T$, the mapping returns a finite subset from the set R of tuples with common schema E .*

A data stream can also be described as an ordered sequence of elements evolving in time. Its current state may include all tuples accumulated so far. Furthermore, an instance of the stream at any specific time instant is the finite set of tuples with that distinct timestamp value. In general, all of the above definitions can be generalized for multiple streams of data.

2.2 Online Joins over Data Streams

The online stream join operator applies a specified predicate among tuples coming from two different stream relations. In most cases and due to the unbounded nature of data streams, this kind of operator is applied over portions of the most recent tuples, referred to as *windows*. Nevertheless, this is not necessarily true for several systems, which also support this operator over full or partial-historical states of the stream [3] [22].

In general, different types of windows [10] may be specified over data streams. The most common flavor of windows is the time-based sliding window. This kind of window is defined by means of time units. A time-based window of WS time units contains all tuples $\{t \mid t'.t_s - t.t_s \leq WS\}$, where t' is the latest received tuple in the respective stream. In this respect, we may provide a formal definition of the online time-based windowed join over two streaming relations [10]:

Definition 4 (Online Windowed Join): *The online windowed join is a binary operator that may be applied between two streaming relations. The windows for each relation may be of the same or different types and scopes. At each time instant $t \in T$, the windowed join between two streams returns the concatenation of pairs of tuples which match a predicate condition, taken from either window state.*

In particular, we intend to join tuples from two logical stream S_1 and S_2 . Whenever the predicate $P(t_{s1}, t_{s2})$ holds for tuples $t_{s1} \in S_1$ and $t_{s2} \in S_2$, an output tuple t_o is produced combining t_{s1} and t_{s2} and appropriately setting the timestamp. The predicate condition involves attributes from both streams (e.g. $S_1.A_i = S_2.A_j$). Additionally, several policies may be adopted for the timestamp of the newly created output tuple. For example, the most recent timestamp value can be chosen, as a way to preserve ordering in the derived stream. An alternative solution would be to attach the minimum between the two original timestamp values, with the interpretation that the output tuple should expire as soon as one of the original tuple expires.

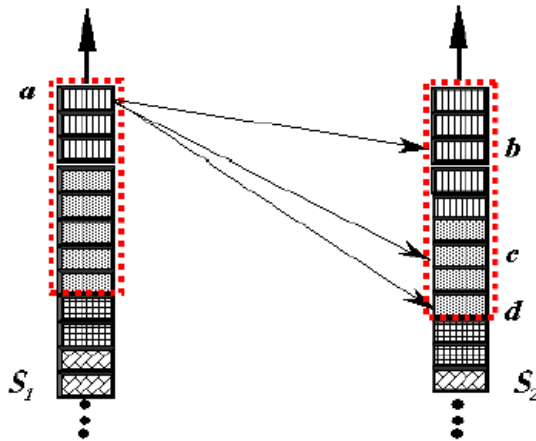


Figure 1: Windowed Join [10]

Figure 1 depicts an online join operation between two data streaming relations S_1 and S_2 with different windows specified over each of them. Each incoming tuple in a given relation is tested for possible matching of the predicate condition with every tuple in the opposite relation in the designated window. Arrows show potential matches to be returned [10].

2.3 Related Work on Parallel Stream Joins

Much research effort has been conducted on parallel stream join processing. These works can be classified into two main categories as either shared-nothing or shared-memory models/techniques. Regarding shared-memory models, they focus on joining infinite data streams in the context of multi-core and main-memory environments. To name a few shared-memory algorithms, Handshake-Join [6] is one of the earliest proposals. This model organizes the processing units (threads or processes) as a doubly-linked list and the incoming streaming relations are directed into the system from opposite sides. The join predicate is evaluated once the relation tuples meet in some processing unit. Intuitively, this model reminds the way that football players from opposite teams exchange handshakes prior to game beginning. However, this model may be sensitive regarding message loss and node failure when moved to the distributed environment. The Hells-join has also been proposed recently [33]. This operator exploits novel properties of state of the art processors. CellJoin [7] algorithm parallelizes stream joins on multi-core Cell processors. Its effectiveness relies heavily on the parallelization techniques of the underlying hardware. One of the latest proposals for shared-memory systems is ScaleJoin [1]. Its architecture relies on the underlying ScaleGate structure which stores the streaming data in a non-blocking concurrent skip-list. This technique demonstrates lower latency, better throughput and linear scalability with respect to the underlying hardware threads when compared with Hand-Shake Join. This kind of operator is not a good fit for a transition to an elastic distributed environment, because of the strong data dependencies among its processing units. A great deal of research work has also been made on shared-nothing parallel joins in a distributed environment with a cluster of commodity machines. For example, Photon [11] is designed for joining data streams of web search queries (click-stream analytics) in Google. This system supports only equi-joins. Chakraborty and Singh [4] present a technique for parallelizing windowed stream joins over a shared-nothing cluster with controversial results. D-Streams [31] decomposes continuous streams into discrete batches and processes them on Apache Spark [30]. This kind join processing may only provide approximate results, as a few target tuple pairs in different batches may miss each other for join operation even if they match the join predicate. TimeStream [34] is another paradigm of distributed stream join processing system. Similar to our approach, this system offloads the computation to a cloud provider. It exploits the dependencies of tuples to perform the joins. However, TimeStream incurs high communication overhead to maintain the dependencies or synchronize the distributed join states. In similar spirit, Elseidy et al [22] present an adaptive operator for shared-nothing parallel joins, but in a data flow setting that does not consider sliding windows (our focus in this thesis). Furthermore, the presented operator adopts the join-matrix stream join model [32]. This model was presented over a decade ago, but has been recently revisited for some systems to support distributed join-processing. It organizes the processing units as a matrix, with each cell holding partitions of both relations. However, this model suffers from high memory consumption, because it presents high replication requirements. Each incoming tuple has to be replicated among multiple processing units. Additionally, scaling operations are not trivial to implement because of the difficult maintenance of the matrix structure. The join-biclique model has been introduced by Qian Lin et al [3] in order to overcome both deficiencies and for that purpose we opted to adopt this model for our project.

2.4 The join-biclique model

The *join-biclique* stream join model was introduced by Qian Lin et al in the 2015 ACM SIGMOD International Conference on Management of Data [3]. The key design goals of *join-biclique* are:

- The facilitation of scalability in multi-node environments.
- The mitigation of memory requirements in the overall distributed streaming join system.

Following the above requirements, *join-biclique* is modeled after the complete bipartite graph or biclique. Thus, we initially present the formal definition of a biclique from graph theory [35]:

Definition 5 (Complete bipartite graph): *In the mathematical field of graph theory, a complete bipartite graph or biclique is a special flavor of bipartite graph where every vertex of the first set is connected to every vertex of the second set.*

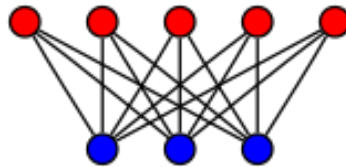


Figure 2: Complete Bipartite Graph

Figure 2 depicts a complete bipartite graph or biclique. It is a graph whose vertices are partitioned into two distinct subsets V_1, V_2 , such that no edge has both endpoints belonging in the same subset. Every possible edge that could connect two vertices in different subsets is part of the graph. That is, it is a bipartite graph (V_1, V_2, E) for which the following holds: For every two vertices $v_1 \in V_1$ and $v_2 \in V_2$, there exists a distinct edge $v_1 v_2$ in E . A complete bipartite graph with partitions of size $|V_1|=m$ and $|V_2|=n$, is termed $K_{m,n}$. Every two graphs with the same notation are isomorphic.

The join-biclique model is based on the biclique graph for joining data streams. In order to proceed with the formal definition of the model, we should firstly present the table (Table 1) that lists the main symbols along with the corresponding descriptions used in the definition and throughout the rest of this report. Afterwards, the model definition follows.

Table 1: Join-biclique symbols

Symbol	Description
R, S	Streaming Relations
r, s	Streaming Tuple
R_i, S_j	i – th and j – th partition of R and S
G_R, G_S	Complete set of partitions of R and S
$G_{R,k}, G_{S,l}$	Subgroup of partitions of R and S
m, n	Number of partitions of R and S
d, e	Number of subgroups of R and S
W_s	Size of the sliding window
P	Archive period of the chained in – memory index

Definition 6 (Join-Biclique Model): Given a cluster of $n+m$ processing units, the *join-biclique* model organizes them as a complete bipartite graph. Supposing there are two streaming relations R and S , the processing units of relation R belong exclusively to the first subset of the bipartite graph for storage and similarly the processing units of relation S belong to the second subset of the graph. That is, each subset of the graph corresponds to one of the relations for storage. Particularly, tuples from relation R are partitioned and stored into one subset of the graph with n units and without replication and tuples from relation S are similarly partitioned and stored into the opposite subset of the graph with m units. The complete set of partitions that belong to R can be defined as the universal set of the n processing units holding the relation's data: $G_R = \{R_1, R_2, \dots, R_n\}$. The same holds for S : $G_S = \{S_1, S_2, \dots, S_m\}$. In the biclique graph, for every two processing units (vertices) R_i and S_j , there exists a distinct edge $r_i s_j$, where $i \in \{1, 2, \dots, n\}$ and $j \in \{1, 2, \dots, m\}$. Every distinct edge represents a potential join result produced by $R_i \bowtie S_j$.

Each edge of the graph in the *join-biclique* model represents the join operation between two units of the opposite relations. This model is capable of generating the Cartesian product of the joinable tuples and thus it supports any kind of join predicate. By and large, the basic dataflow setting of the model is the following: Upon receiving an incoming tuple, *join-biclique* always stores it on exactly one processing unit without data replicas, and produces the output by sending the tuple to all the machines that (may) contain joinable tuples from the opposite relation. One property of the model is that all processing units are independent of each other. This property is very important for our system because it also allows us to create a collection of small isolated services (the processing units), each of which owns their data and is independently isolated, scalable

and resilient to failure. Thus, we can easily define a microservices-based architecture from this kind of model, which will be flexible, scalable and elastic [5]. A typical deployment may not necessarily operate in a distributed setting. That is, one physical machine may host several processing units (services) and thus the relations R and S may be only logically separated.

Figure 3(b) presents a possible join-biclique model organization. The relation R is split into two ($n = 2$) partitions (R_1, R_2) and relation S is split into three ($m = 3$) partitions (S_1, S_2, S_3). The result of $R \bowtie S$ may be obtained if we join every $R_i \bowtie S_j$ with R_i being the i -th partition of R and S_j being the j -th of S , where $i \in \{1, 2, \dots, n\}$ and $j \in \{1, 2, \dots, m\}$.

2.4.1 Comparison with Join-Matrix Model

As already mentioned in section 2.3, the join-matrix model [32] has been recently revisited [22] as an alternative way of organizing the processing units of a distributed join processing system. In particular, this model organizes the units as a matrix, with each axis corresponding to one of the two relations. This is the case for the 2-way joins that we are examining in this project. In a scenario with multi-way joins, the matrix may be represented as a Hypercube [23]. The join-matrix scheme distributes the incoming tuples on the axes of the originating relation and replicates on the other axes. For example, when $r_1 \in R$ arrives in the system, it may be directed to one partition of R (e.g. R_1). This partition may consist of several processing units and the incoming tuple will be replicated to all of them. In the meantime, it is joined with every partition of S stored in these units.

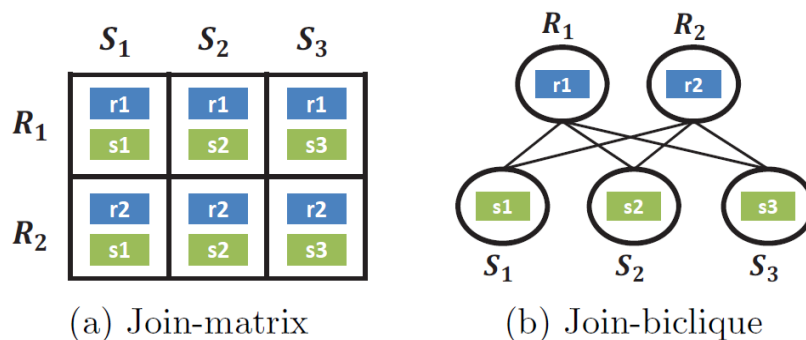


Figure 3: Stream Join Models [3]

Figure 3(a) depicts how the join-matrix model is organized. This model is able to handle 2-way stream theta-joins in a distributed, parallel and decentralized manner. However, it is not amenable to scaling because of matrix dependencies and also suffers from high memory consumption because of the replication requirements. To this end, join-biclique model was introduced by Qian-Lin et al. in order to address both issues. However, such advantages may pose specific shortcomings under certain circumstances. For example, join-biclique poses higher network communication cost than join-matrix when using random partitioning. If we compare the two models assuming that the relations are of equal sizes, each relation in join-biclique uses $\frac{p}{2}$ processing units. On the other hand, the join-matrix model is represented by a $\sqrt{p} \times \sqrt{p}$ matrix. On the former model, each

tuple has to be sent to $\frac{p}{2}$ processing units for the joining operation, while on the latter model each tuple is sent only to \sqrt{p} processing units.

3. ELASTIC-BICLIQUE SYSTEM ARCHITECTURE

We designed and developed an alternative implementation of a distributed stream join system based on join-biclique model. Our system is able to scale dynamically on-demand and is built using novel streaming technologies and tools, such as RabbitMQ and the Spring Cloud Stream framework [37]. We are focusing on windowed-based stream joins and specifically time-based sliding windows [10], as in many streaming scenarios [1] [38] [39].

3.1 System Design

The elastic-biclique system is built using a microservices-based architecture; it is a fully-fledged distributed system, which comprises from a collection of small, isolated services, each of which holds a specific role in the environment of operation. Each service owns its data and is independently isolated from other services as well as scalable and resilient to failures. Different types of services integrate with other types to form a flexible and cohesive distributed system.

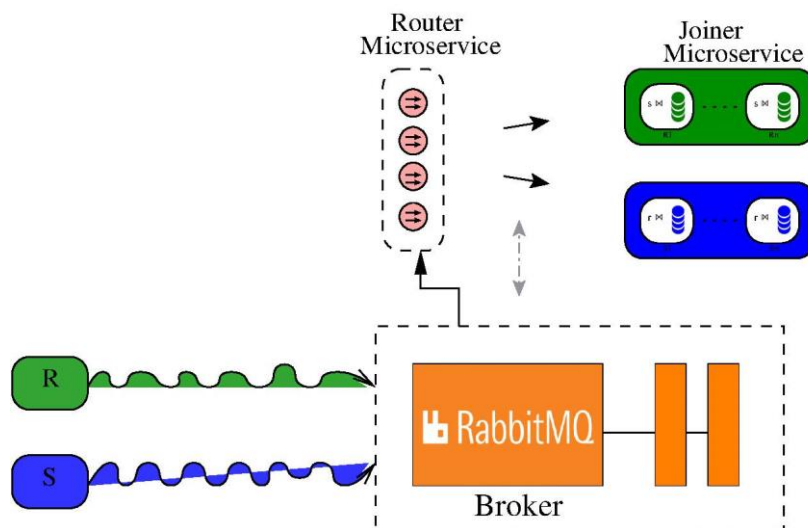


Figure 4: Overall architecture of elastic-biclique

Figure 4 presents a high-level overview of the various services and systems involved along with the interconnections between them. The main functional components of elastic-biclique consist of the **router** and **joiner** microservices along with the **RabbitMQ** message broker [41]. We adopted a message-driven microservices architecture. That is, the microservices rely on asynchronous message passing for inter-communication. This kind of communication is necessary in order to decouple them, and their communication flow both in:

- **Time:** to better facilitate concurrency
- **Space:** to allow distribution and mobility

Without this decoupling it is impossible to reach the level of compartmentalization and containment needed for isolation and resilience [5]; properties of great importance for the elastic-biclique system. The RabbitMQ message broker was employed to implement this kind of asynchronous intercommunication among the microservices. In summary, the main services involved in the architecture are the following:

- **stream-service:** This service acts as a stream source adapter which emits streaming relations into the system.
- **router-service:** This service acts as a dispatcher, which ingests incoming tuples and routes them to corresponding services.
- **joiner-service:** This service consists of all the processing units. It is responsible for join-processing.
- **message-broker:** This service is responsible for ingesting and retransmitting streaming tuples to the microservices.

Our system is implemented along the lines of the join-biclique model but tries to reduce the inter-unit connectivity among its services compared to the original model. The same idea is implemented by Qian Lin et al. for their system BiStream. Intuitively, each processing unit in the join-biclique model needs to connect with every unit in the opposite relation. Fortunately, this kind of inter-communication among the services isn't necessary if we separate data routing and data joining procedures between different services. That is, each joiner service (from each relation) only communicates with the router service to receive the data to either store or join without directly communicating with each other. More specifically, imagine an incoming tuple $r \in R$ from an external source arriving at the system. Initially, it enters a router service which directs it to one of the services R_i for storage (in the respective time-based sliding window) and at the same time is sent to all the units of S for join processing. After the join processing, r can be discarded from all the units in S .

3.1.1 Router

The router service is designed to ingest the incoming tuples from the input streaming relations and direct them to the corresponding joiner units for further processing. Except from the ingestion of tuples and routing decision tasks, the router is also responsible for maintaining statistics related to input data, such as rate of events per second. The message broker is involved both for ingesting the input tuples and directing them back. Different channels, exchanges and queues inside the broker are used to achieve the above functionality. More details about the specific implementation of input/output channels within the broker may be found in Chapter 4.

3.1.2 Joiner

The joiner service ingests the incoming tuples from the router. It comprises all the processing units of the distributed join processing system. The joiner services are separated into the two main subsets of the bipartite graph and they can be viewed as parallel partitions of the two streaming relations (R, S) . These services serve two main purposes which translate into two different execution branches. The first branch is responsible for data storage (of the tuples which are of the same type of relation as the service and belong to the current time-based sliding window) and the other for join processing of tuples belonging to the opposite relation. The join processing involves the join predicate comparison and the stale tuple discarding from the time-based sliding window. To put it into perspective, imagine a tuple $r \in R$ arriving at its corresponding joiner service $R_i \in G_R$ for storage. It will be added in the current time-based sliding window. If a tuple $s \in S$ arrives in the same R_i unit, it proceeds with a pairwise

comparison of the join predicate with all the tuples stored in the designated time-based sliding window. Furthermore, the invalidation of stale tuples from the window should be performed. The steps involved in this procedure will be described below. We should also note that the data discarding operation is important for releasing memory. This process is implemented following the theorem from [3]:

Theorem 1: *The stored tuples $r \in R_i$ can be safely removed from the current time-based sliding window, when R_i receives an incoming tuple $s \in S$, such that $s.t_s - r.t_s > W_s$. In similar spirit, $s \in S$ can be discarded from once S_j receives one tuple such that $r.t_s - s.t_s > W_s$.*

Refer [3] for the corresponding proof of the above theorem. We also need to support different in-memory indices, in order to index and efficiently join process the tuples from both relations based on join predicate. We use a HashMap for equi-join and a BinarySearchTree for non-equi-join predicates. As the authors in [3] point out, it is not efficient to organize the entire streaming data with one single index, as it will incur high overhead during the stale tuple discarding operation. In order to overcome this kind of overhead, we adopt their idea in our implementation, named chained in-memory index. Figure 5 depicts a schematic structure of this model.

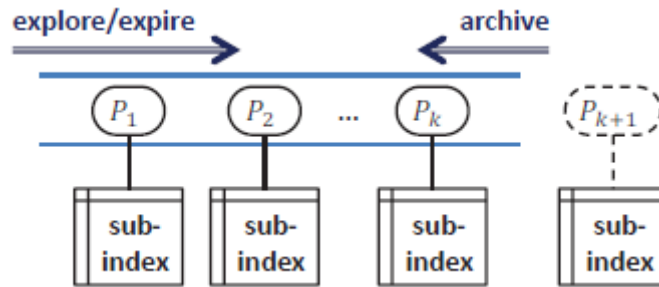


Figure 5: Chained in-memory index [3]

The main idea of the chained in-memory index model is to partition the streaming tuples based on the discrete time intervals and construct a sub-index per interval. Each sub-index is associated with the minimum and maximum timestamps of the tuples that it holds. This time interval is named the archive period P . The sub-indices are chained together as a linked-list ordered by the construction time of each index. The stale tuple discarding operation can be now performed in the context of the sub-index level rather than the tuple level and thus we reduce the overhead of the operation, since the valid sub-indices are not affected when the obsolete sub-indices are discarded. In general, the following basic operations are performed by the joiner services with respect to the time-based sliding window constraint and the chained in-memory index model:

- **Data Indexing:** When an input tuple arrives at a joiner for storage, it will be first added into the active sub-index and update the min/max timestamps. Next, it will calculate the difference between min and max timestamps and if this value exceeds the designated archive period P , then the current active sub-index will become inactive and archived into the chain and a new empty active sub-index will be created. Otherwise, the current active sub-index will remain active.
- **Data Discarding:** An inactive sub-index may become expired and removed from memory by dereferencing it. An inactive sub-index becomes expired when a tuple reaches the joiner service and belongs to the opposite relation. According to

Theorem 1, if the difference between the timestamp of current tuple and the maximum timestamp of the sub-index in question is larger than W_s , then the sub-index will be expired. Thus, we improve efficiency by avoiding the pairwise comparison between every tuple inside each sub-index.

- **Join Processing:** After marking the expired sub-indices, an incoming tuple from the opposite relation needs to join with all the tuples in the remaining sub-indices (both the current active and archived). A pairwise window comparison is performed with all tuples belonging in these sub-indices.

3.1.3 RabbitMQ Broker

RabbitMQ is a complete open-source broker implementation of the Advanced Message Queuing Protocol (AMQP) [42]. It started as a joint project of LShift and CohesiveFT in 2007. It is written in the Erlang programming language and is built using the Open Telecom Platform (OTP); Erlang’s framework for clustering and failover [43]. In particular, RabbitMQ is implemented as an extra AMQP layer on top of OTP using Erlang, thus benefiting from the robustness, reliability and flexibility of a proven platform. Since 2013, RabbitMQ is part of Pivotal Software. Among the key benefits of RabbitMQ are the following:

- High reliability
- High availability
- Scalability
- Good throughput and latency performance
- Extensive management and monitoring control
- Debugging facilities
- Implementations of tooling and clients in various programming languages (e.g. Java, Python etc)

3.1.3.1 AMQP Protocol

In this subsection, we will briefly present some of the basic functionality of the AMQP protocol as defined in the specification manual [45]. AMQP was initially presented in 2003 by John O’Hara at JPMorgan Chase in London, UK [44]. It is an application layer protocol (on top of a reliable transport layer protocol e.g. TCP) for message-oriented middleware. The latter entails software or hardware infrastructure which supports sending and receiving messages between distributed systems. Particularly, AMQP creates full functional interoperability between clients and messaging middleware servers (also termed “brokers”). AMQP provides flow controlled, message-oriented communication with message-delivery guarantees, along with encryption and authentication. Both the networking protocol and the server-side semantics are sufficiently defined through:

- The Advanced Message Queuing Protocol Model (AMQ model), which defines a set of components that direct and store the messages within the broker, plus a set of rules for wiring these components together.

- A wire-level protocol (AMQP) which enables clients to directly interact with the broker. A wire-level protocol refers to a way of moving data from point to point in a network.

Regarding the AMQ model, it specifies the following main types of components, which are interconnected in various ways inside a broker service:

- **Exchange:** This is the main entry point inside the broker from the outside world. It receives messages from producer applications and routes them to different messages queues based on prearranged criteria, such as message contents or properties. The model defines two different types of exchanges:
 - **Direct:** It routes based on a routing key
 - **Topic:** It routes on a routing pattern
- **Message queue:** This is where incoming messages are stored until one or many consumer application(s) processes them.
- **Binding:** This abstraction defines the relationship between a message queue and an exchange and provides the message routing criteria to different queues.

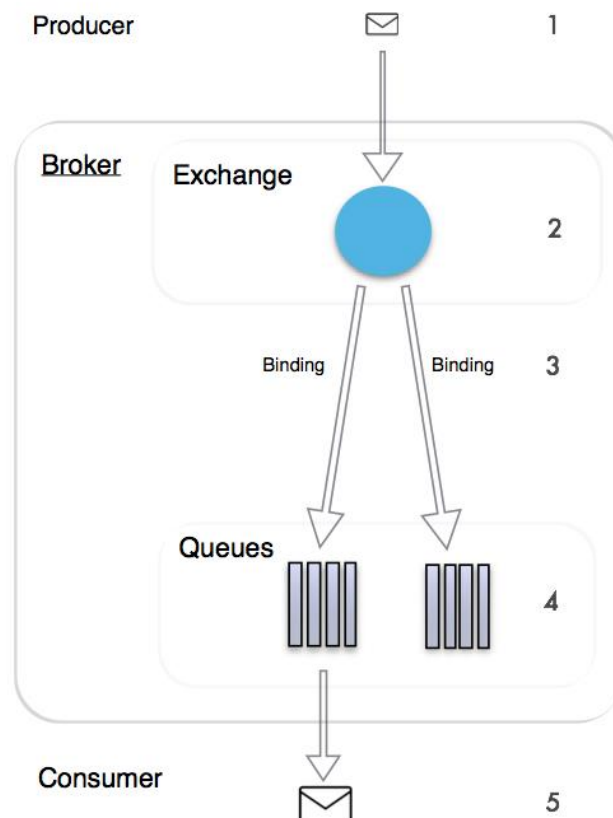


Figure 6: Message Flow in AMQ model

Figure 7 depicts a complete message flow in the AMQ model which entails all the functional components. A typical flow would be the following:

1. The producer application publishes a message to an exchange.
2. The exchange receives the message and is now responsible for the routing of the message. The exchange takes different types of message attributes into account, such as routing key or routing pattern.
3. Bindings have to be created from the exchange to queues. In this case we see two bindings to two different queues from the exchange. The Exchange routes the message in to the queues depending on message attributes.
4. The messages stay in the queue until they are handled by a consumer.
5. The consumer consumes/processes the message.

Regarding, the wire-level (AMQP) protocol, it is characterized by the following features: (i) multi-channel, (ii) asynchronous, (iii) secure, (iv) portable and (v) efficient. Figure 6 depicts the two layers in which AMQP is split.

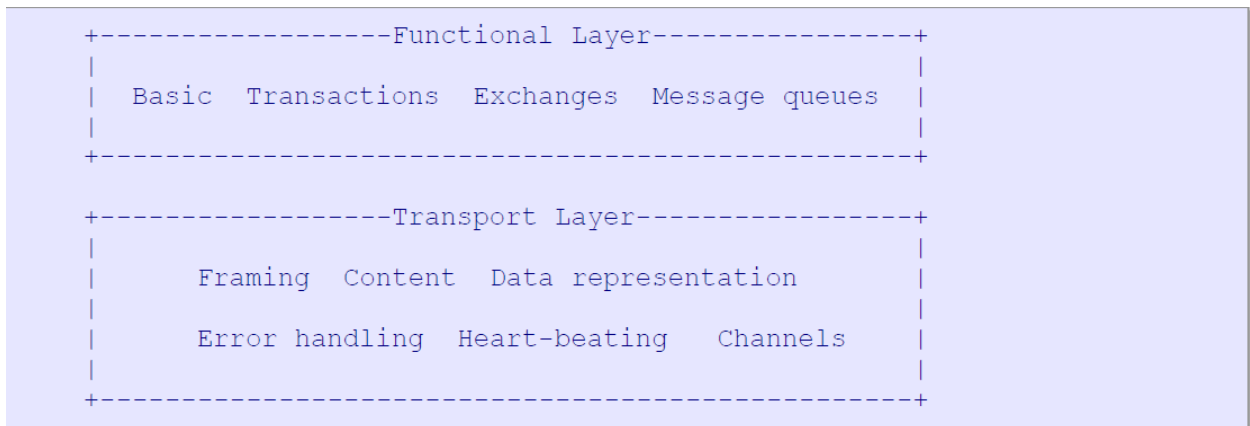


Figure 7: AMQP Layers [45]

The functional layer uses a set of different commands, which are grouped into logical classes of functionality. These commands perform useful work on behalf of the application. The transport layer is responsible for the bidirectional transfer of these methods from the application to the broker (and backwards) [45]. It handles framing, content encoding, heart-beating, error handling, data representation and channel multiplexing.

3.2 Dataflow Control

Our main purpose of controlling the dataflow is to balance the load among the different services of the system and at the same time the efficient join processing of the two data streams. Tuples from both streams are initially entering the system in the RabbitMQ broker, where they are direct through a dedicated topic exchange. A single message queue is bound to that exchange and receives all the incoming tuples. A pool of router services (consumers) reads the tuples and each tuple goes to one of them (randomly). In messaging systems, this kind of model is usually termed queuing model. The strength of this model is that it allows the system to divide up the processing of data over multiple consumer instances and thus enable effective load balancing. This fact also lets us scale our processing (related to the router services). For implementation details of the above strategy, please refer to Chapter 4.

Once, a tuple enters a router service it is segregated into two different streams: the store stream and the join stream. The store stream routes each tuple to a join service for storage and similarly the join stream routes each tuple to the proper join services for join processing. We implemented different routing strategies for these two streams based on join selectivity.

For high-selectivity predicates that involve anything but equality-joins, a random routing strategy is adopted. In particular, non-equi-join (high-selectivity) predicates may generate a large number of results over most of the join processing elements. A random routing strategy should be preferred for this kind of predicates, because it ensures equal load balancing among the processing units of a relation and protects from load imbalance when the data is skew. For example, in a non-equi-join predicate scenario, let $r \in R$ entering a router service. It is randomly routed to one unit in R for storage without taking into account the contents of r via the store stream. In addition, r is sent to all units in S for join processing via the join stream. The opposite is true for a tuple $s \in S$. The store and join streams again involve the RabbitMQ message broker. For the store stream, a topic exchange with a single message queue is created. All the joiner services from a single relation are competing for the tuples. This technique ensures load-balancing among the units of the relation in a round robin fashion. It is implemented in the same way that was described in the previous paragraph for both streaming relations that initially enter the system and consumed by the router services. More implementation details should be found in Chapter 4. For the join stream, a dedicated topic exchange is created and then a number of queues is bound to that exchange. Each queue corresponds to a joiner service from a relation, from which the service consumes the tuples. In this way, we can achieve that all units in the target relation receive the tuples in the join stream as required from the random routing strategy. This model for distributing the join stream reminds us of the publish-subscribe model as each tuple is broadcast to every unit in the relation.

However, the random routing strategy poses high network communication cost along with extra needless processing cost, because each tuple from the opposite relation has to be sent to all the units in the current relation for join processing. For low-selectivity joins such as equi-joins, we can alleviate these costs and achieve more efficient join processing by implementing a hash-partitioning routing strategy. In particular, a hash-partitioning routing strategy, routes the tuples based on the hash value of the join attribute and targets them into a specific unit. Tuples with the same hash value on the join attribute end into the same unit. This technique helps us to perform efficient join processing by guaranteeing data locality. We can implement this strategy by hashing of the join attribute and enforcing the relevant topic exchanges to route tuples to specific

queues (that match the hash value) for both store and join streams. Implementation details of this strategy should be found in Chapter 4.

3.3 Tuple Ordering Protocol

We should implement a protocol that ensures the reliability of join results at the level of the join processing services. Faulty join results may arise if tuples arrive out-of-order from the store and join streams. This kind of disorder in data streams may arise from many sources, such as stream items being routed by different paths in a network, or combining streams that are out of synch [47]. We follow a protocol design along the lines of the BiStream system as presented in [3].

Before presenting the protocol semantics we should identify the possible scenarios that may produce error-prone join results. Figure 7 presents all the possible orders that two tuples r and s may reach the join processing services S_j and R_i .

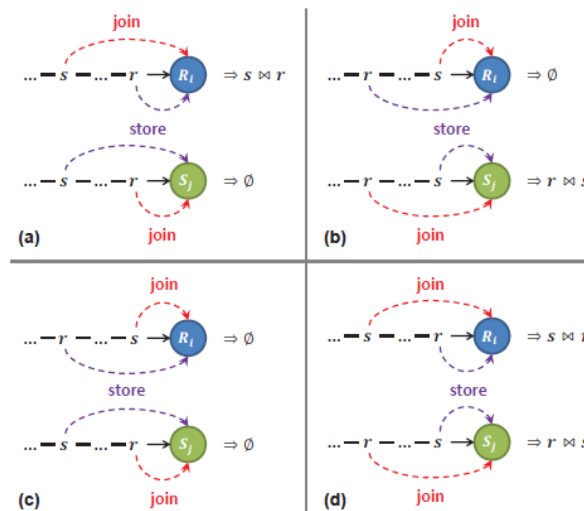


Figure 8: Arrival order of tuples r and s . [3]

In Figure 8(a) the r tuple is stored in R_i before the s tuple arrives in the same service for join processing. Then the R_i produces a single join result. At the same time, the r tuple arrives in S_j for join processing earlier than the s tuple (for storage) and thus is discarded and no join result is produced. This case produces a single correct join result between r and s , because the result is produced exactly once in R_i . Figure 8(b) depicts a symmetric scenario as in 8(a), where a single join result is produced. Figures 8(c) and 8(d) present faulty scenarios. In 8(c) we can observe that a missed join result occurs, when no join result is produced in R_i , because s arrives for join processing earlier than r arrives for storage and thus is discarded. At the same time, no join result is produced in S_j , because r arrives out-of-order for join processing earlier than s arrives for storage. In 8(d), we can observe that a duplicate join result occurs due to out-of-order arrivals of tuples. In particular, a join result is produced in R_i , when r arrives for storage earlier than s arrives for join processing, thus producing a result. The same join result is produced in S_j , when when r arrives for join processing later than s arrives for storage.

In order to avoid these two possible error cases, we should consider implementing a protocol, which guarantees the join results by processing the tuples in consistent order

at the joiner service level. Qian Lin et al in [3] present two definitions of an order consistent protocol along with a pairwise FIFO protocol, which may be the guide for a possible implementation. This implementation should provide guarantees about processing the tuples from the store and join streams in consistent order at the joiner services. The definitions are presented as follows:

Definition 7 (Order-Consistent Protocol): *Given a set of router services Y and a set of join processing services U , each router service $y_i \in Y$ sends a set of tuples $X_i = \{x_{i1}, \dots, x_{ik}\}$ as a stream. Each tuple is broadcast to a set of joiner services. A network protocol is called order-consistent if and only if: There exists a global tuple sequence $Z = \{x_{z1}, \dots, x_{zk}\}$, where Z contains each tuple exactly once. For each unit $u_j \in U$, it receives all the tuples assigned to it (i.e., no loss in the network), and the sequence of tuples it processes is a subsequence of the global tuple sequence Z .*

The above protocol is presented in [3] to ensure that the relative order for any two joining tuples r and s only depends on a single global order and is consistent over all services. The following pairwise FIFO protocol is similarly presented to guarantee that for every pair of router and joiner services the in-between message passing and processing is FIFO.

Definition 8 (Pairwise FIFO Protocol): *Given a router service y and a joiner service u , y sends a set of tuples $X = \{x_1, \dots, x_k, \dots\}$ to u as a stream. A network protocol is called pairwise FIFO if and only if: For any two tuples x_a and x_b , if x_a is sent by y before x_b , then x_a is processed at u before x_b .*

The key idea presented with the above protocol is that every tuple holds a counter or id that is incremented by one after every sent tuple by the router. The joiner only processes those tuples with the expected id. When a tuple is delayed or lost, the processing of the following tuples stops and should be continued only after receiving the tuple with the expected id or perform some kind of synchronization action. Different flavors of this protocol may be implemented.

The authors of [3] present an implementation of the order-consistent protocol based on the pairwise FIFO protocol. We opted to follow this protocol for our implementation in order to address the two faulty aforementioned scenarios of tuple disorder in the streams. The protocol is implemented as follows: Each tuple receives a monotonically increasing counter at each router service. This counter is incremented by one for every tuple sent by the router. Each joiner service maintains and sorts the incoming tuples based on the counter in question; the global sequence referenced in Definition 7 is preserved on the order of the counter. Additionally, the joiner should be aware of the appropriate timing to proceed with processing the currently maintained and sorted tuples. The router should somehow signal the joiner that it should proceed with the sorted tuples. A stream punctuation technique [47] is used to implement this kind of intervention from the router to the joiner. A punctuation is a pattern p inserted into the data stream with the meaning that no data tuple t matching p will occur further on in the stream. In our case, every router service emits a signal tuple with a counter to all the joiner services periodically (e.g. every 20ms). Recall that message passing between every router and joiner service is FIFO. Such a signal tuple indicates that all tuples (from this router) with counter less than the signal counter have been received by the joiners and thus the joiners should proceed with processing them. The joiners maintain a priority queue for tuples coming from both the store and join streams and proceed with processing the tuples that have smaller counter than the latest received signal counter.

The latest counters from all the routers are stored in a table in each joiner service and get update periodically. With this kind of protocol, we are able to guarantee proper ordering of the sent tuples at the joiner services and thus ensure join results completeness.

4. SYSTEM IMPLEMENTATION

In this chapter, we are going to describe various aspects of the implementation of the elastic-biclique system. As we already mentioned, this system is an alternative implementation of the join-biclique model presented in [3]. We opted for an implementation using the concept of message-driven microservices. Our development was carried out in the Java programming language using the open-source Spring Boot [18] and Spring Cloud Stream [19] frameworks.

4.1 Spring Boot

The Spring framework is a very popular Java-based framework for building web and enterprise applications. It exists since around 2003 and its main abstractions are dependency injection (DI) and inversion of control (IoC). Unlike many other frameworks, which focus on only one area, Spring framework provides various modern features addressing the modern business needs via its portfolio projects. At its very core, Spring framework bases its functionality on the concept of beans; objects that form the backbone of an application and are managed by the IoC component of Spring [18]. In other words, a bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. The framework itself provides the flexibility to configure beans in multiple ways, e.g. XML, Annotations, and JavaConfig. With the number of features increased the complexity also gets increased and configuring Spring applications becomes tedious and error-prone. To this end, the Spring Boot project is the next-generation attempt at easy Spring setup configuration. This project makes it easy to create Spring-based stand-alone applications that one can easily deploy and run with minimum effort on configuration aspects. The primary goals of this project as stated in [18] are:

- Provide a radically faster and widely accessible getting started experience for all Spring development.
- Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
- Absolutely no code generation and no requirement for XML configuration.

While Spring Boot provides the foundation for creating DevOps friendly microservice applications, other libraries in the Spring ecosystem help create Stream based microservice applications. The most important of these is Spring Cloud Stream.

4.2 Spring Cloud Stream

Spring Cloud Stream builds on Spring Boot to create stand-alone Spring applications. In particular, it is a framework for building message-driven microservice applications. It also uses Spring Integration [48] to provide connectivity to message brokers [19] (including RabbitMQ). Other features and concepts of this framework include:

- Opinionated configuration of several middleware messaging brokers (Kafka, RabbitMQ, Redis).
- Persistent publish-subscribe concepts.
- Consumer groups

- Partitions

From this point-of-view, it seems like the perfect framework for our system. It allows us to easily build stream-based microservice applications without dealing with low-level complexity and also offers us out-of-the-box integration with the RabbitMQ broker. We continue this chapter by describing the main concepts of this framework.

4.2.1 Main Concepts

The essence of the Spring Cloud Stream programming model is to provide an easy way to describe multiple input and output channels of an application that communicate over a messaging middleware. Specifically for our case, those input and outputs map into RabbitMQ exchanges and queues. Common application configuration for a Source that generates data, a Process that consumes and produces data and a Sink that consumes data is provided as part of the library.

In essence, a Spring Cloud Stream application consists of a neutral middleware core. The deployed application communicates with the outside world through input and output channels injected into it by the framework. These channels connect to the external broker. Figure 9 presents an overview of the above concept.

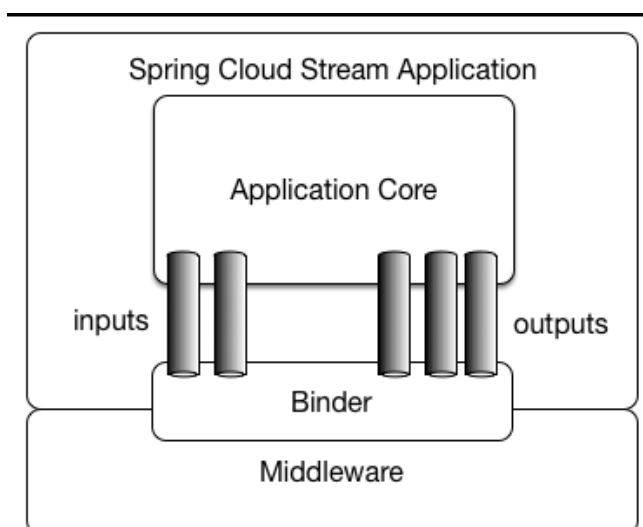


Figure 9: Spring Cloud Stream abstractions [19]

As of April 2017, the framework provides Binder implementations for RabbitMQ, Kafka and Redis. It automatically detects and uses a binder found on the classpath.

As we have already described, messaging has two models: queuing and publish-subscribe. In the queuing model, a pool of consumers may read from a producer and each data item goes to one of them. On the contrary, in the publish-subscribe model the data items are broadcast to all of the consumers. Each model has a strength and a weakness. The strength of queuing is that it allows dividing up the processing of data over multiple consumer instances, which lets us scale up. Unfortunately, queues aren't multisubscriber; once a consumer reads the data item then it's gone. The strength of

publish-subscribe is that it allows us to broadcast data to multiple processes, but has no way of scaling processing since every message goes to every subscriber.

Spring Cloud Stream generalizes the above two concepts. Particularly, the communication between different microservices follows a publish-subscribe model, where data is broadcast through shared topics. The publish-subscribe communication model reduces the complexity of both the producer and the consumer, and allows new applications to be added to the ecosystem without disrupting the existing flow. Added to that, the queuing model is achieved through the concept of *consumer groups* (Inspired by Apache Kafka consumer groups). All groups which subscribe to a given destination receive a copy of published data, but only one member of each group receives a given message from that destination. By default, when a group is not specified, Spring Cloud Stream assigns the application to an anonymous and independent single-member consumer group that is in a publish-subscribe relationship with all other consumer groups. Figure 10 illustrates the concept of the consumer groups in Spring Cloud Stream.

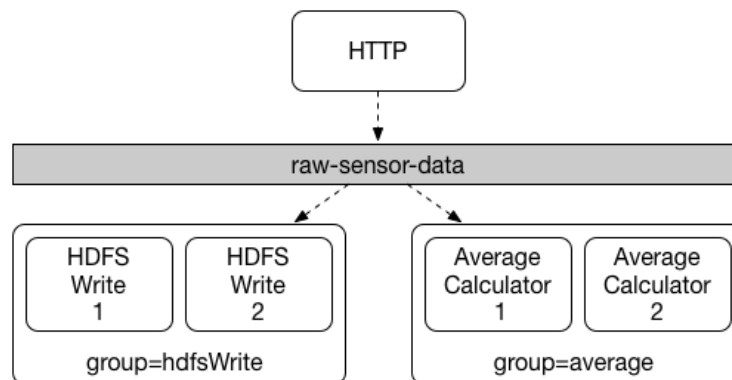


Figure 10: Concept of Consumer Group [19]

Other important concepts of Spring Cloud Stream are Durability and Partitioning support. The former allows for consumer groups subscriptions to be durable. This means that the binder implementation ensures that when a subscription for a group is created, it automatically becomes durable meaning that the group will receive messages even if they are sent while all applications in the group are stopped. The latter feature (partitioning support) enables support for partitioning data between multiple instances of a given application. In a partitioned scenario, the broker exchange (in Rabbit) is viewed as being structured into multiple partitions. One or more producer applications send data items to multiple consumer application instances and ensure that items presenting common characteristics are processed by the same consumer instance. Figure 11 illustrates the concept of partitioning. Several producer applications send via HTTP, data items to the same topic (exchange) which is partitioned. Items with common characteristics are directed to a specific partition. Only a unique Average Processor application instance consumes these data and thus we ensure that they are processed together.

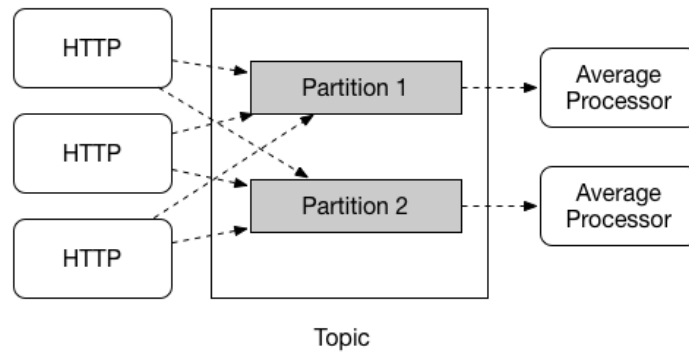


Figure 11: Partitioning Concept [19]

Even though RabbitMQ broker does not support physical exchange partitioning like Kafka, Spring Cloud Stream offers a common abstraction for implementing partitioning in a uniform fashion. Partitioning support is very critical for our system, so we could implement hash partitioning in the joiners for efficient equi-join processing.

4.3 Implementation Analysis

Based on the abstractions of Spring Cloud Stream, we designed and developed the elastic-biclique system. When using the RabbitMQ binder, each destination is mapped to a Topic Exchange. Figure 12 depicts the binder.

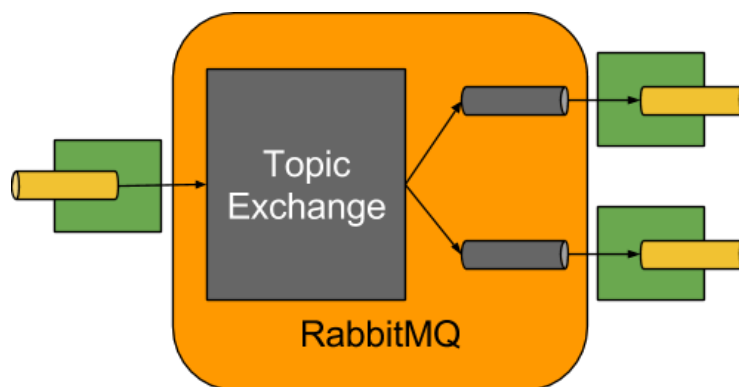


Figure 12: RabbitMQ binder [19]

For every consumer group, a queue is bound to that topic exchange. Each consumer instance has a corresponding RabbitMQ consumer instance for its group's queue. That is, consumer instances that belong to the same group are competing with each other for the data tuples. For partitioned producers/consumers the queues are suffixed with the partition index and use the partition index as the routing key.

Regarding our implementation, the entry point for the tuples of both streaming relations (R, S) is a topic exchange named *tuple.exchange*. A single queue is bound to that exchange which corresponds to the consumer group of the router instances. Using this abstraction, the router instances compete with each other for the ingestion of the incoming tuples from both relations. This way we can also easily scale up or down the router-services depending on the tuple rate. Each router instance is responsible for

directing the tuples to the corresponding joiner services for further processing. We will provide a dataflow example for a tuple from relation R in order to demonstrate the implementation. The corresponding procedure applies for tuples coming from the S relation (with S -store and S -join exchanges).

Imagine a tuple coming from R relation, it should be directed to the R-joiner services for storage and at the same time to the S-joiner services for join processing. For that purpose, tuples from R are directed to an R -store topic exchange for storage by R-joiner instances and to an R -join exchange for join processing by S-joiner instances. If the random routing strategy is adopted concerning high-selectivity joins (Section 3.2), a single queue is bound to R -store exchange and all R-joiner instances (the belong to the same consumer group) compete for the tuple; it ends up to one of them. For the same routing strategy, multiple queues are bound to the R -join exchange, each corresponding to one instance of the S-joiners (they don't belong to a specific consumer group thus multiple queues are created and bound to the exchange). This way, the tuple is broadcast to all of them for join processing in a publish-subscribe fashion.

If we adopt the hash partitioning strategy for low-selectivity joins (equi-joins), multiple queues are bound to the R -store topic exchange, each corresponding to a single R-joiner instance. A tuple from R is directed to a single queue based on the hash value of the joining attribute and thus a single R-joiner instance consumes and stores the tuple. Similarly, multiple queues are bound to the R -join topic exchange, each corresponding to a single S-joiner instance. The same tuple from R is directed to a single queue based on the hash value of the joining attribute and thus a single S-joiner instance consumes and join-processes the tuple. Using this strategy, we ensure that tuples from both relations having the same hash values on the joining attribute, will end up in the same instance.

As we already mentioned, using Spring Cloud Stream we can easily describe the multiple input and output channels of our microservice instances. These channels are communication channels to/from the RabbitMQ broker. To better understand the coding style (using Annotations) of Spring Cloud Stream, we will provide the router service's channels implementation interfaces:

```
public interface TupleSink {
    String CHANNEL_NAME = "tuplesChannel";

    @Input(TupleSink.CHANNEL_NAME)
    SubscribableChannel tuplesChannel();
}
```

The `@Input` annotation identifies an input channel, through which received messages enter the application. It can also take a channel name as a parameter. In our case for the router service, the channel name is called `tuplesChannel` and defines the input communication channel from which the router service receives tuples from both relations. Similarly, we define an interface for the output channels:

```
public interface TupleSource {

    String CHANNEL_NAME_R_STORE = "tuplesChannelRstore";
    String CHANNEL_NAME_R_JOIN = "tuplesChannelRjoin";
    String CHANNEL_NAME_S_STORE = "tuplesChannelSstore";
    String CHANNEL_NAME_S_JOIN = "tuplesChannelSjoin";
}
```

```

    @Output(TupleSource.CHANNEL_NAME_R_STORE)
    MessageChannel tuplesChannelRstore();

    @Output(TupleSource.CHANNEL_NAME_R_JOIN)
    MessageChannel tuplesChannelRjoin();

    @Output(TupleSource.CHANNEL_NAME_S_STORE)
    MessageChannel tuplesChannelSstore();

    @Output(TupleSource.CHANNEL_NAME_S_JOIN)
    MessageChannel tuplesChannelSjoin();
}

```

The `@Output` annotation identifies an output channel, through which published messages leave the application. In our case, we define 4 different output channels in the router service; two channels with store, join semantics per relation. These channels have appropriate naming.

4.4 Docker containers

Docker is the world's leading container platform. In particular, it is a tool designed to make it easier to create, deploy and run applications by using software containers. Containers are fundamentally based on a feature of the Linux kernel named Namespaces. This feature allows the isolation and virtualization of system resources of a collection of processes. Examples of resources that can be virtualized include process IDs, hostnames, user IDs, network access, inter-process communication, and filesystems. Furthermore, containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By using containers, resources can be isolated, services restricted, and processes provisioned to have an almost completely private view of the operating system with their own process ID space, file system structure, and network interfaces. Multiple containers share the same kernel, but each container can be constrained to only use a defined amount of resources such as CPU, memory and I/O [20].

Docker containers rely on docker images to run. In particular, a Docker image is the template (application plus required binaries and libraries) needed to build a running Docker Container (the running instance of that image). Each Docker image references a list of read-only layers that represent filesystem differences. Layers are stacked on top of each other to form a base for a container's root filesystem. When we create a new container, we add a new, thin, writable layer on top of the underlying stack. This layer is often called the "container layer". All changes made to the running container - such as writing new files, modifying existing files, and deleting files - are written to this thin writable container layer [20].

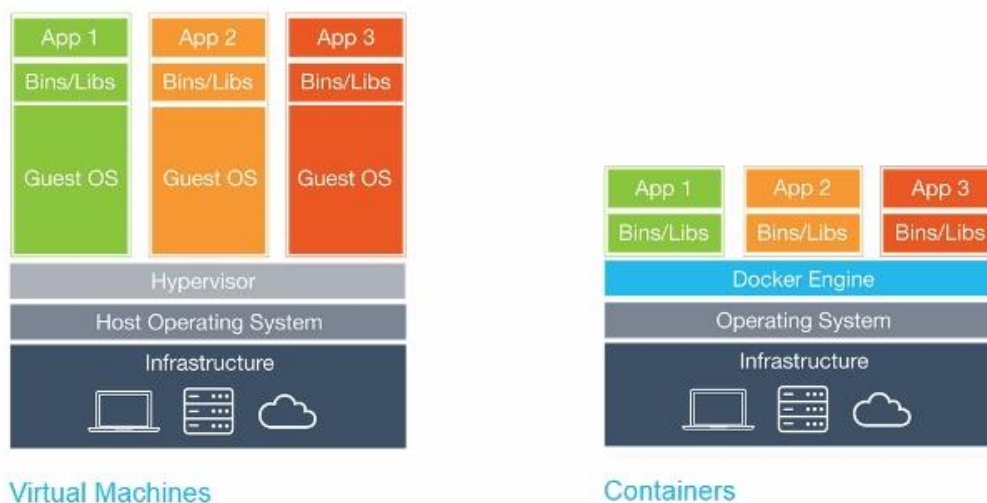


Figure 13: VM vs Containers

In a way, Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications to be shipped with things not already running on the host computer. Figure 13 depicts this kind of model. Using this abstraction we can achieve a significant performance boost and reduce the size of an application. Using Docker to create and manage containers may also simplify the creation of highly distributed systems by allowing multiple applications, worker tasks and other processes to run autonomously on a single physical machine or across multiple virtual machines. This latter characteristic is very important for our distributed elastic-biclique system. Additionally, docker containers allow efficient scaling of an application by spawning new containers of a given service on demand. The opposite procedure (scale down) is equally easy, thus allowing us to use the resources only when we need it.

In our container cluster, we are going to use three different Docker images. The first image corresponds to the RabbitMQ broker, while the second and the third images will correspond to the router and joiner services respectively. These latter images are based on `alpine-oraclejdk8`, which holds the Alpine Linux distribution along with the OracleJDK 8, totaling only ~170MB in size.

4.5 Kubernetes

Container orchestration is one of the hottest topics in industry. Initially, the industry focused on pushing container adoption. The next step is forward is to put containers in production at scale. There are many tools in this area. Some examples are Apache Mesos, Docker Swarm and Amazon's ECS and Kubernetes. For this thesis, we will focus on Kubernetes; an open-source container orchestration system meant to be deployed on Docker-capable clustered environments. Currently, it is one of the fastest-moving open source projects and seems to be winning the competition. Statistics on GitHub prove this fact: Kubernetes is in the top 0.01 percent in stars and No. 1 in terms of activity.

This system was initially developed at Google and its name originates from the greek word «Κυβερνήτης» meaning governor or commander. It is commonly abbreviated as "k8s", which is derived by replacing the 8 letters "ubernete" with 8. Kubernetes's technology isn't precisely new. Behind the open source community uptake hide

exceptional engineering efforts. That is, 15 years of Google's active development and heavy production for a product named Borg; the cluster management tool that powered the infrastructure behind Gmail, YouTube, Google Search, and other popular Google services. Kubernetes's success relies on 15-plus years of Google R&D that goes into Borg's code.

Kubernetes provides several features such as grouping, load-balancing, auto-healing, scaling, autoscaling, container replication, volume management, infrastructure monitoring, rolling updates, service discovery, identity, authorization etc. The framework distinguishes the participating nodes between master and worker nodes. The master provides a unified view into the cluster and, through its publicly-accessible endpoint, is the doorway for interacting with the cluster. The worker(s) are managed from the master, and run the services necessary to support Docker containers. Each node runs the Docker runtime and hosts a Kubelet agent, which manages the Docker containers scheduled on the host. Each node also runs a simple network proxy. Additionally, the way Kubernetes functions is by using pods that group into containers, then scheduling and deploying them at the same time. While most other container management services use a container as their minimum unit, Kubernetes uses the pods. A pod is a group of one or more containers, the shared storage for those containers, and options about how to run the containers. Pods are always co-located and co-scheduled, and run in a shared context. Generally, a pod contains one or more application containers which are relatively tightly coupled. These pods are quickly updated, built, or destroyed in real-time depending on the situation. Kubernetes can be used on private, public, multi-cloud, and hybrid cloud environments. With all these features and convenient abstractions, Kubernetes seemed like the perfect orchestration framework for our containerized distributed system.

4.6 Google Container Engine

Google Container Engine (GKE) is a powerful cluster manager and orchestration system [13] for running Docker containers. Particularly, it is built on top of Google Compute Engine, which is a typical public Infrastructure as a Service (IaaS) cloud platform, similar to Amazon Elastic Compute Cloud [12], Microsoft Azure [14] or OpenStack [49]. The engine schedules Docker containers into the cluster and manages them dynamically based on predefined requirements (such as CPU and RAM). The main benefit of GKE is that it is built on top of Kubernetes, giving us the flexibility to take advantage of the public cloud infrastructure.

Using GKE, we are able to set up a container cluster along with the orchestration framework within minutes. We do not have to deal with the strenuous and non-trivial process of configuring the Kubernetes framework. Thus, we are able to direct our focus on the elastic-biclique containerized system deployment and execution on the cloud.

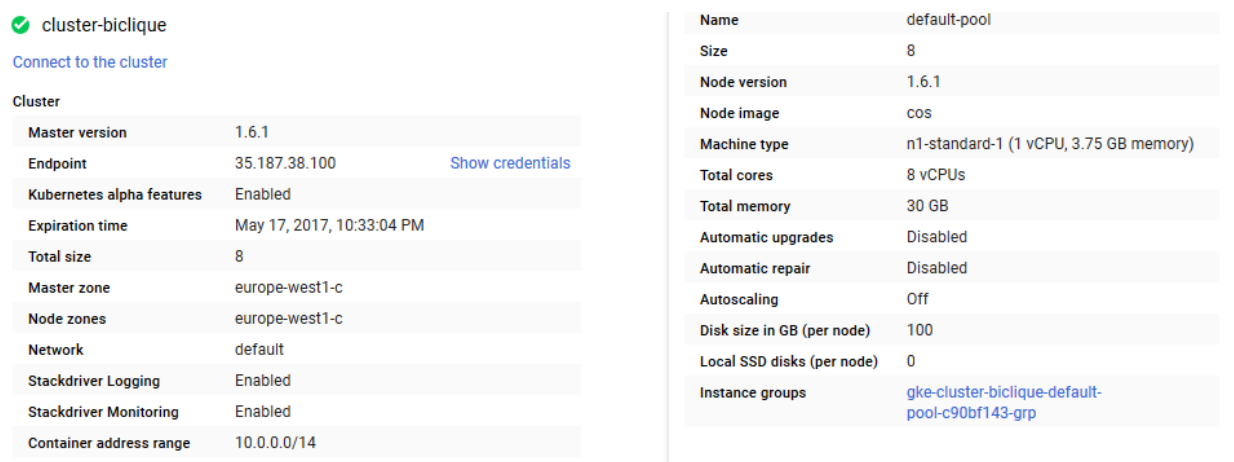
Furthermore, GKE offers a flexible auto-scaling feature that helps optimize resource efficiency. The auto-scaling refers both to containers (pods) within particular VMs (worker nodes) and also to VM instances. The former feature is offered by Kubernetes, while the latter feature is offered cluster auto-scaling by Google Compute Engine (IaaS). Our main focus on this project is to demonstrate elasticity of the distributed stream join system and as such the Kubernetes pod auto-scaling feature is of great importance.

5. DEPLOYMENT AND EXPERIMENTS

5.1 Setup and Deployment

We deployed our elastic-biclique system on Google Container Engine. Our system comprises of three core Docker images, namely the rabbitmq-broker, router and joiner services. RabbitMQ is already available on Docker-Hub and we chose to use the smallest available image (rabbitmq:alpine) with a size of ~5MB. The other images are also based on the Alpine Linux distribution along with the OracleJDK 8 with a size of ~170MB each. We pushed these images on the public Docker-Hub registry under the eangelog/\$service-name-service tag. Furthermore, we used single container Kubernetes pods based on our images. A single container pod has only one container running inside it. This way, we could ideally run each container on a separate VM instance.

Google Container Engine (GKE) is a cluster manager and orchestration system based on the public IaaS cloud of Google (Google Compute Engine). As with every public cloud provider, GKE comes with specific Service Level Agreement (SLA) constraints and billing requirements for leasing VM instances. In order to conduct the experiments required by this thesis project and due to limited monetary resources, we opted to use the free-tier infrastructure offered by GKE. This tier is free of charge and is offered for a limited amount of time (12 months) and also with a limited amount of monetary resources (300 US dollars). Unfortunately, the free-tier comes also with a set of pre-defined resource quotas per account. For example, a maximum of 8 CPU cores per zone and a maximum of 100 images are enforced (among others). Due to such resource limitations (especially the 8 CPU cores quota), our experiments were significantly limited. We expect to conduct more sophisticated experiments in the future, when we will have access to different IaaS providers.



The screenshot displays the configuration details for a GKE cluster named 'cluster-biclique'. It is divided into two main sections: 'Cluster' and 'Node pool'.

Cluster	
Master version	1.6.1
Endpoint	35.187.38.100 Show credentials
Kubernetes alpha features	Enabled
Expiration time	May 17, 2017, 10:33:04 PM
Total size	8
Master zone	europa-west1-c
Node zones	europa-west1-c
Network	default
Stackdriver Logging	Enabled
Stackdriver Monitoring	Enabled
Container address range	10.0.0.0/14

Node pool	
Name	default-pool
Size	8
Node version	1.6.1
Node image	cos
Machine type	n1-standard-1 (1 vCPU, 3.75 GB memory)
Total cores	8 vCPUs
Total memory	30 GB
Automatic upgrades	Disabled
Automatic repair	Disabled
Autoscaling	Off
Disk size in GB (per node)	100
Local SSD disks (per node)	0
Instance groups	gke-cluster-biclique-default-pool-c90bf143-grp

Figure 14: Cluster information

We created a cluster in GKE named cluster-biclique with an initial size of 8 VMs. Figure 14 summarizes the cluster information on GKE. Each VM has an Intel Xeon vCPU @ 2.5 GHz and 3.75GB of RAM along with 100GB of ephemeral local disk. The total compute resources are 8vCPUs, 30GB of RAM and 800GB of disk size. The 8 VMs refer directly to Kubernetes worker nodes, as GKE abstracts away the Kubernetes master node from the cloud client. The cluster default region is europa-west and the zone is 1-c. The Kubernetes version is 1.6.1 and we chose to turn off the cluster auto-scaler, because of our limited resource constraints. This feature is also currently in a

beta version. Instead, we will focus on the Horizontal Pod Autoscaler (HPA) of Kubernetes for our experiments.

We initiated the bootstrap of our cluster with the command depicted in Figure 15.

```

root@eangelog-P67A-UD3:/home/eangelog/kube_deployments# gcloud alpha container clusters create cluster-biclique --cluster-version=1.6.1 --machine-type=n1-standard-1 --num-nodes=8 --enable-kub
This will create a cluster with all Kubernetes Alpha features enabled.
- This cluster will not be covered by the Container Engine SLA and should
  not be used for production workloads.
- You will not be able to upgrade the master or nodes.
- The cluster will be deleted after 30 days.

Do you want to continue (Y/n)? Y
Creating cluster cluster-biclique...done.
Created [https://container.googleapis.com/v1/projects/elastic-biclique/zones/europe-west1-c/clusters/cluster-biclique].
kubeconfig entry generated for cluster-biclique.
NAME          ZONE          MASTER_VERSION  MASTER_IP      MACHINE_TYPE  NODE_VERSION  NUM_NODES  STATUS
cluster-biclique europe-west1-c  1.6.1 ALPHA (29 days left)  35.187.38.100  n1-standard-1  1.6.1      8      RUNNING
root@eangelog-P67A-UD3:/home/eangelog/kube_deployments#

```

Figure 15: Cluster bootstrap

We have to note here that we chose to deploy the cluster with all the Kubernetes alpha features enabled. The so-called Alpha cluster is a short-lived cluster that is not covered by the Container Engine SLA and cannot be upgraded, but has all Kubernetes APIs and features enabled. This kind of cluster is a way to run stable Kubernetes releases with Alpha features that may be less stable. These clusters are automatically destroyed after 30 days. We needed the alpha features enabled, so we could be able to use the HPA's alpha feature of auto-scaling based on the resource metric of memory. This feature may be found in the `autoscaling/v2alpha1` API.

After our cluster is bootstrapped, it is time to deploy our containers comprising the elastic-biclique system. Kubernetes offers several abstractions which makes scaling and managing containers a facile task. We consider the most important of these abstractions to be: (i) Pods, (ii) Services and (iii) Deployments. We have already given a brief explanation of Pods in section 4.5. On the other hand, Services is a core abstraction of Kubernetes, which provides persistent endpoints for Pods. In particular, Pods aren't meant to be persistent. They can be stopped or started for many reasons and this leads to communication problems, because restarted Pods may have different IP addresses. Finally, Deployments are a declarative way to ensure that the number of Pods running is equal to the desired number of Pods, specified by the user. The main benefit of Deployments is in abstracting away the low level details of managing Pods. Behind the scenes, Deployments use Replica Sets (another core abstraction of K8s) to manage starting and stopping the Pods. If Pods need to be updated or scaled, the Deployment will handle that. Deployments also handle restarting Pods if they happen to go down for some reason. All the above abstractions could be described with YAML files, which are sent to the Kubernetes API server.

In general, we chose the Deployment abstraction to deploy our images into our biclique-cluster, in order to benefit from all the handy features offered by this abstraction. The first step of the deployment procedure of elastic-biclique is the creation of a Deployment for the RabbitMQ broker, along with two Services; the first Service is used for the internal communication of other biclique services with the broker at port 5672 and the second service provides external access to RabbitMQ GUI through port 15672. Figure 16 demonstrates the above Services.

```

root@eangelog-P67A-UD3:/home/eangelog/kube_deployments# kubectl get services
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kubernetes    10.3.240.1      <none>           443/TCP          9m
rabbitmq      10.3.249.77     <none>           5672/TCP         2m
rabbitmq-mgmt 10.3.242.40     146.148.112.213 15672:32678/TCP  2m

```

Figure 16: Kubernetes Services

For the router and joiner services we also created two different Deployments. For that purpose, we wrote a YAML file per Deployment. For example, the following snippet describes a possible router-service Deployment:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: biclique-router
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: biclique-router
    spec:
      containers:
        - name: biclique-router
          image: "eangelog/router-service"
          env:
            - name: SPRING_APPLICATION_JSON
              value: '{"spring.rabbitmq.addresses": "rabbitmq"}'
```

The above Deployment uses the extensions/v1beta1 API and creates a Replica Set to bring up 2 router-service Pods using the image eangelog/router-service. We also use an environment variable named spring.rabbitmq.addresses=rabbitmq, which will be used by the containerized router Java application to point to our RabbitMQ broker service. A DNS service is provided by Kubernetes on GKE for discovering registered Services.

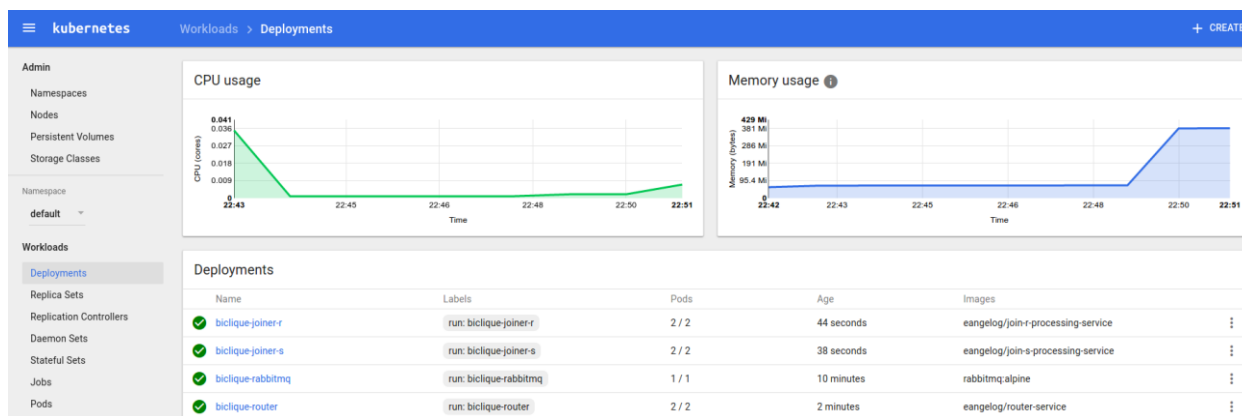


Figure 17: Kubernetes Deployments

Figure 17 illustrates the successful deployment of the core services of elastic-biclique (router, joiner and rabbitmq) as Deployments from the Kubernetes Dashboard, along with CPU and memory usage. At this particular moment, there is no incoming stream traffic and the Pods remain idle. We should note that there are two starting Pods per type of joiner (R, S) and router and one Pod for RabbitMQ (we do not seek High-Availability for the queues of the broker for these experiments). Figure 18 presents the various queues involved in the biclique system (section 4.3 for details) from the RabbitMQ management GUI. For these experiments we adopted the random routing strategy (section 3.2 for details) and as such the corresponding number/type of queues are bound to the R, S (store, join) exchanges for the two Pods of each relation.



Queues

▼ All queues (7)

Pagination

Page 1 of 1 - Filter: Regex (?)(?)

Overview			Messages			Message rates			+/-
Name	Features	State	Ready	Unacked	Total	Incoming	deliver / get	ack	
Rjoin.exchange.anonymous.7Xxrz9AERieL4SsK5ktKuQ	AD	idle	0	0	0				
Rjoin.exchange.anonymous.tOrckMNRSI-xMpPBtdIgAQ	AD	idle	0	0	0				
Rstore.exchange.Rstoregroup	D	idle	0	0	0				
Sjoin.exchange.anonymous.MTHqsswITAW-vHVbAIVxtg	AD	idle	0	0	0				
Sjoin.exchange.anonymous.YjuF5bPnTB2TcAC8_CUJHW	AD	idle	0	0	0				
Sstore.exchange.Sstoregroup	D	idle	0	0	0				
tuple.exchange.routergroup	D	idle	0	0	0				

Figure 18: RabbitMQ idle queues

Now that our elastic-biclique services are deployed into the GKE cluster, we can proceed with executing the auto-scaling experiments.

5.2 Experiments

We now need to demonstrate the capability of the elastic-biclique system to dynamically adjust the number of Pods according to stream input rate changes. For that purpose, we used the Horizontal Pod Autoscaler (HPA) of Kubernetes; a feature which enables Kubernetes to automatically scale the number of pods in our Deployments based on observed CPU utilization (or, with alpha support, on some other, application-provided metrics such as memory). We need to provide a brief description of the technical aspects of HPA before continuing with the experiments.

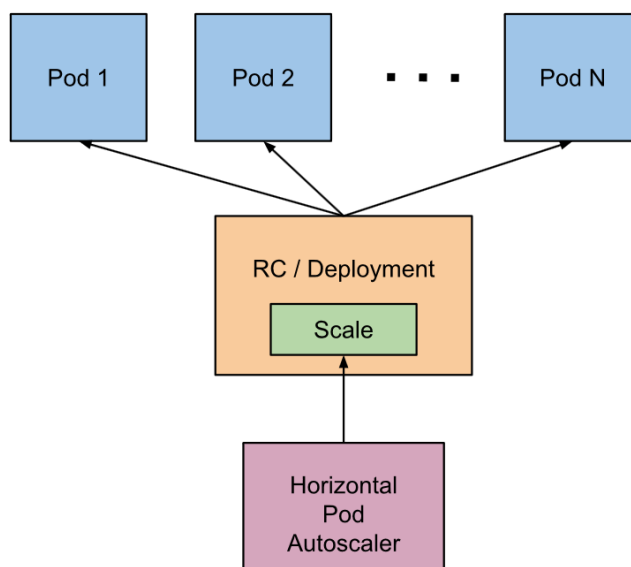


Figure 19: Horizontal Pod Autoscaler

Figure 19 provides an overview of HPA. The Horizontal Pod Autoscaler is implemented as a control loop with a predefined period of operation (e.g. 30 seconds). During each period, the controller manager queries the resource utilization against the metrics specified in each HPA definition. The controller manager obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics). For these experiments we are interested in per-Pod resource metrics (CPU and memory). For this kind of metrics, the controller fetches the values from the resource metrics API for each pod targeted by the HPA. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted pods, and produces a ratio used to scale the number of desired replicas. The HPA controller can fetch metrics in two different ways: direct Heapster access, and REST client access. In our case, we use the direct Heapster access as it is the default way when deploying GKE clusters. Heapster is an open-source project which enables container cluster monitoring and performance analysis. When using direct Heapster access, the HPA queries Heapster directly through the API server's service proxy subresource. Heapster needs to be deployed on the cluster and running in the kube-system namespace [50].

We conducted our experiments with HPA based on two different resource metrics, namely CPU utilization and memory load. Due to resource and time constraints, we only evaluated a single equi-join query for a 10-minute window join in 60 minutes of duration. Our cluster resource constraints refer to the small number of vCPU cores available (8), which limits both the possible tuple input rate and the number of Pods that we are able to deploy on our cluster.

The autoscaling operation based on the CPU utilization may also be described as a YAML file which is submitted to the Kubernetes api-server:

```
apiVersion: autoscaling/v2alpha1
kind: HorizontalPodAutoscaler
metadata:
  name: biclique-joiner-r
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: biclique-joiner-r
  minReplicas: 1
  maxReplicas: 3
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 80
```

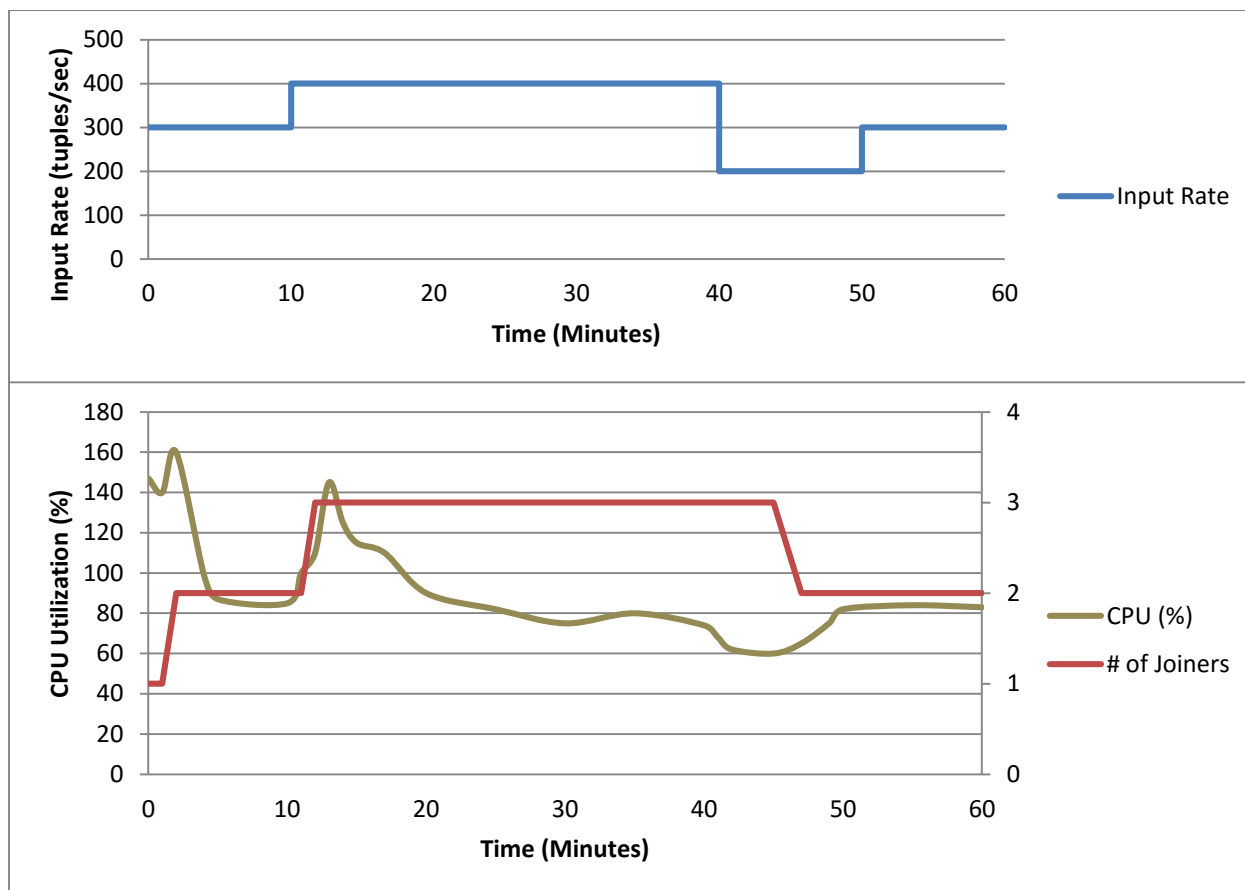


Figure 20: Dynamic Scaling based on CPU utilization

Figure 20 presents the varying stream input rates during the 60-minute of our evaluation. The upper part of Figure 20 shows the stream input rate, while the bottom part shows how the joiner Pods are dynamically added/released from the system while the CPU utilization changes. We set the target CPU utilization value at 80% and the minimum and maximum amount of Pods at 1 and 3 respectively. These values can be viewed also in the YAML file. We start with a rate of 300 tuples/sec for the first 10 minutes and a single joiner per relation. The initial CPU utilization is far above the desired target value at ~145%, so a second joiner Pod launches by the autoscaler. Following this action, the utilization seems to stabilize for the next 10 minutes below the 80% target. At the 10th minute, we increase suddenly the rate to 400 tuples/sec and the utilization also rises at a constant rate. The autoscaler decides to bootstrap a third joiner Pod to balance the load. The utilization seems to balance again around the target value, over the next 30 minutes until the 40th minute of our evaluation. At the 40th minute, we decrease the input rate at 200 tuples/sec and as such we can observe a decrease in the utilization below 60% with 3 Joiner Pods. Thus, the autoscaler decides to decrease the number of Pods to 2 again. At the 50th minute, we increase the input rate again at 300 tuples/sec and as such we can observe a stabilization of the utilization again at around 80%.

Figure 21 presents the auto-scaling experiment based on memory load. Before describing the results depicted, we consider important to report the technique that we used for optimal behavior, regarding memory footprint of the Java Virtual Machine (JVM) in cloud operation. If the JVM is run using the default, parallel GC with no configuration flags provided, other than the heap maximum (-Xmx), JVM will try to use all the available heap right up to that maximum. It keeps allocating new data out of the available address space until it runs out. Only then does it collect all the live data and

compact it down into the bottom of the heap, before continuing to fill up the free space and so on. That's true even when the application would run perfectly happily in much less space. In cases where a single machine is dedicated to the JVM that's not necessarily a problem. But in Cloud deployments like elastic-biclique, many JVMs are deployed as virtualized guests (containers) sharing the resources of an underlying host machine. Clearly, when there is competition for memory it is preferable for the JVM to use as small a memory footprint as is compatible with keeping down memory management costs. A JVM can easily monitor how much live data an application is holding on to. If this is much lower than the configured heap maximum, then garbage collection and compaction can be performed early, before all the heap space is filled. That allows each JVM to unmap the unused address space at the top end of the heap, making more physical memory available for other JVMs. The gain is that you can either run more JVMs on the same physical host or run the same number of JVMs on a similar host installed with less memory. Both options translate to saved money.

Luckily, we can alleviate the above problem by using proper configuration of the JVM, as it already implements a memory footprint management policy. Using this policy we can force our JVM to keep the mapped heap space fairly close to the application's live data set size. Using the following JVM flags in our Joiners we can achieve the aforementioned behavior:

```
-XX:UseParallelGC
-XX:MinHeapFreeRatio=20
-XX:MaxHeapFreeRatio=40
-XX:GCTimeRatio=4
-XX:AdaptiveSizePolicyWeight=90
```

At any GC the collector can decide to map more of the available heap pages into the nursery space or mature space. It can also decide to unmap pages and work in less space. The footprint control model makes mapping decisions based upon the values of two parameters, `MinHeapFreeRatio` and `MaxHeapFreeRatio`, with default values 40 and 70. These two heap ratios specify what percentage excess memory should be mapped beyond that occupied by the live set. Let's assume, for simplicity, that at GC the live objects occupy 100Mb. The defaults specify that the mapped pages should lie between 140Mb and 170Mb. If the currently mapped heap space is less than 140Mb it needs to be extended by mapping more physical pages. If it is more than 170b it needs to be trimmed by unmapping pages. Obviously, these limits are themselves constrained by the heap minimum and maximum supplied on the java command line (`-Xms` and `-Xmx` settings).

Our configuration resets the heap ratios to 20 and 40. This makes the GC to trim the extra heap space much more tightly, keeping it much closer to the live data set size. So, with 100Mb of live data the heap would be adjusted to lie between 120Mb and 140Mb, i.e. – there would be about half as much excess space. If the application's live set size and allocation rate remain constant then this means that GCs would have to happen about twice as often with these settings.

The time goal is configured by two parameters, `GCTimeRatio` and `AdaptiveSizePolicyWeight`, with default values 99 and 10. `GCTimeRatio` specifies the worst case GC time the collector should target. A value of 99 means no more than 1% of time should be spent in GC. In practice, that means that the parallel GC has to play cautious. So, it regularly trades off space for time even when the actual GC time is a tiny fraction of 1%. When a young GC occurs it tends just to add more heap, ignoring the `MaxFreeHeapRatio` value. The result is that the heap size just keeps rising, often up to the heap maximum. Our configuration resets `GCTimeRatio` to 4, i.e. a worst case

goal of 20%. This effectively places most of the weight in the competing footprint management goals on space rather than time reduction. With this setting the time goal no longer dominates and the heap stays between the limits defined by `MinFreeHeapRatio` and `MaxFreeHeapRatio`.

The `AdaptiveSizePolicyWeight` parameter controls how much previous GC times are taken into account when checking the timing goal. The default setting, 10, bases the timing goal check 90% on previous GC times and 10% on the current GC time. Resetting this to 90 means that the timing goal check is mostly based on to the current GC execution time, i.e. it is more responsive to current rather than historical memory use. This greater responsiveness also usefully limits the extent to which space gets traded off against time.

We also have to note that we left a default setting for the minimum and maximum heap size per JVM constrained by our physical resources per host. That is, the minimum heap size is 58MBs and the maximum heap size is 926 MBs.

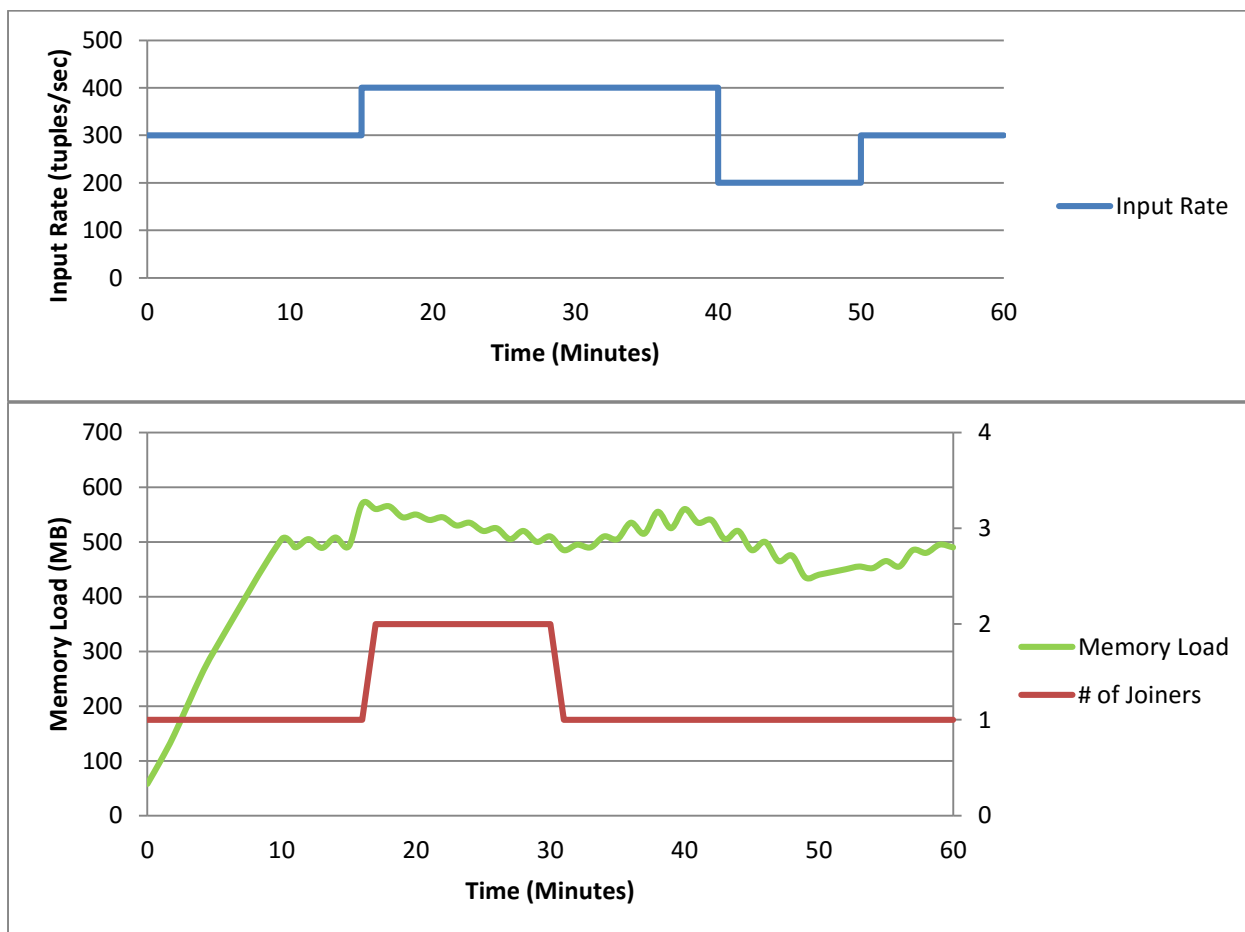


Figure 21: Dynamic Scaling based on Memory Load

We performed the autoscaling experiment based on memory load, using a 10-minute window with 1 Joiner as the default setting. We set the target memory value as 85% of total memory. We adjusted the memory consumption at the host, where the target Joiner Pod resides, so we could hit the target value of 85% at around 520MB of JVM memory. Initially, the experiment begins with the memory load at 60 MB and an average

input rate at 300 tuples/sec. Normally, the memory load would be bound to the size of the workload within the time window, since expired tuples are discarded from memory. We can observe that after a window time, the memory load is bounded via data discarding until the 15th minute at around 500MB. After the 15th minute the input rate rises to 400 tuples/sec and thus we can observe a sudden spike in the memory load, as more and more tuples accumulate inside the time window. Then, the burden of 520MB is violated, so the autoscaler spawns a second Joiner. The rate of tuple accumulation inside the time window is now split between the two Joiners. Thus, we can observe a constant decline in the memory load until the ~30th minute with ~500MB. Then, the autoscaler decides to release the second Joiner. We can again observe a constant rise in the memory load until the 40th minute, when the rate declines to 200 tuples/sec. Then, the memory load declines to almost 420 MB until the 50th minute, when the rate rises again to 300 tuples/sec. From the 50th to the 60th minute the memory load follows again a rising route and we expect it to stabilize at almost 500MB. We should note that during the system scaling, data migration is avoided since the system discards the expired tuples and controls the storage distribution of the new incoming tuples to achieve equivalent load balancing among the Joiners.

6. CONCLUSION

In this report we have adopted the main ideas found in [3], which presented a model for joining streaming data, namely join-biclique. Join-biclique logically models the processing units as a complete bipartite graph for stream joins with no data replication, flexible partition scheme and processing units independence designs. On the basis of join-biclique, we have designed and developed an alternative implementation of the distributed online stream join processing system using modern technologies and design principles, such as software containers and event-driven microservices. Such technologies are best fitted for cloud operation. For our implementation we have used cutting-edge tools, such as Spring Boot, Spring Cloud Stream, Docker containers and Kubernetes. We deployed our system on Google Container Engine (GKE) --an IaaS/PaaS cloud provider-- and demonstrated the feature of dynamic scaling. For our dynamic scaling experiments we have used the Horizontal Pod Autoscaler (HPA) of Kubernetes and two different resource metrics, namely CPU and memory. We showed (at small scale) that the system is able to dynamically adjust the processing units based on variations of the input stream rate.

6.1 Future Work

Using the GKE free trial, we were able to create a small cluster of 8 VCPUs and 15 GB of RAM. As a result, our experiments were conducted in a very resource restricted environment at small scale. In the future, we expect to deploy our system in a larger cluster at a private or public cloud provider (such as OpenStack or AWS) and conduct our experiments at much bigger scale. We are also seeking to use an alternative message broker, such as Apache Kafka. Unlike RabbitMQ, Kafka is able to scale-out on-demand, thus giving us the opportunity to handle much larger input traffic than with RabbitMQ.

ABBREVIATIONS - ACRONYMS

GKE	Google Container Engine
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
IoT	Internet of Things
IoC	Inversion of Control
HPA	Horizontal Pod Autoscaler
AWS	Amazon Web Services
DI	Dependency Injection

REFERENCES

- [1] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafidou, and P. Tsigas, ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join, *IEEE Transactions on Big Data*, vol. pp, issue 99, Nov. 2016, pp. 1-14.
- [2] D. Wampler, *Fast Data Architectures for Streaming Applications*, O'Reilly Media, 2016.
- [3] Qian Lin, Beng Chin Ooi, Zhengkui Wang and Cui Yu, Scalable Distributed Stream Join Processing, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, May 2015, pp. 811-825, Melbourne, Victoria, Australia.
- [4] A. Chakraborty and A. Singh, Parallelizing Windowed Stream Joins in a Shared-Nothing Cluster, *IEEE International Conference on Cluster Computing*, Sep. 2013.
- [5] J. Boner, *Reactive Microservices Architecture*, O'Reilly Media, 2016.
- [6] J. Teubner and R. Mueller, How Soccer Players Would do Stream Joins, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, June 2011, Athens, Greece.
- [7] B. Gedik, R. R. Bordawekar, Philip S. Yu, CellJoin: a parallel stream join operator for the cell processor, *The VLDB Journal — The International Journal on Very Large Data Bases*, vol. 18, issue 2, April 2009, pp. 501-519.
- [8] P. B. Gibbons, Big Data: Scale Down, Scale Up, Scale Out, *IEEE International Parallel and Distributed Processing Symposium*, May 2015, Hyderabad, India.
- [9] J-S Vockler, G. Juve, E. Deelman, M. Rynge and B. Berriman, Experiences Using Cloud Computing for A Scientific Workflow Application, *ACM Proceedings of the 2nd international workshop on Scientific cloud computing*, June 2011, pp. 15-24, San Jose, California, USA.
- [10] K. Patroumpas, T. Sellis, Window Specification over Data Streams, *ACM Proceedings of the 2006 international conference on Current Trends in Database Technology*, March 2006, pp. 445-464, Munich, Germany.
- [11] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, S. Venkataraman, Photon: fault-tolerant and scalable joining of continuous data streams, *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, June 2013, pp. 577-588, New York, New York, USA.
- [12] Amazon Web Services; <https://aws.amazon.com/>. [Accessed Online 26/04/2017]
- [13] Google Cloud Services; <https://cloud.google.com/>. [Accessed Online 26/04/2017]
- [14] Microsoft Azure; <https://azure.microsoft.com/>. [Accessed Online 26/04/2017]
- [15] G. Palis, Cloud Computing: The new frontier of Internet Computing, *IEEE Internet Computing*, Sep. 2010, pp. 70-73.
- [16] BiStream System; <https://www.comp.nus.edu.sg/~dbssystem/bistream/index.html> [Accessed Online 26/04/2017]
- [17] Apache Storm; <http://storm.apache.org/>. [Accessed Online 26/04/2017]
- [18] Spring Boot; <https://projects.spring.io/spring-boot/>. [Accessed Online 26/04/2017]
- [19] Spring Cloud Stream; <https://cloud.spring.io/spring-cloud-stream/>. [Accessed Online 26/04/2017]
- [20] Docker; <https://www.docker.com/>. [Accessed Online 26/04/2017]
- [21] Kubernetes; <https://kubernetes.io/>. [Accessed Online 26/04/2017]

- [22] M. Elseidy, A. Elguindy, A. Vitorovic, C. Koch, Scalable and Adaptive Online Joins, *ACM Proceedings of the VLDB Endowment*, vol. 7, issue 6, Feb 2014, pp. 441-452.
- [23] A. Vitorovic, M. Elseidy, K. Guliyev, K. V. Minh, D. Espino, M. Dashti, Y. Klonatos, C. Koch, Squall: Scalable Real-time Analytics, *Proceedings of the VLDB Endowment*, vol. 9, issue 13, Sep. 2016, pp. 1553-1556.
- [24] V. Stoumpos, A. Delis, Fragment and Replicate Algorithms for Non-Equi-Join Evaluation on Smart Disks, *IEEE International Symposium on Autonomous Decentralized Systems*, March 2009, Athens, Greece.
- [25] A. Okcan, M. Riedewald, Processing Theta-Joins using MapReduce, *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, June 2011, pp. 949-960, Athens, Greece.
- [26] J. W. Stamos, H. C. Young, A symmetric fragment and replicate algorithm for distributed joins, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, issue 12, Dec. 1993, pp. 1345-1354.
- [27] A. Arasu, S. Babu, and J. Widom, The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 2006.
- [28] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, Semantics of Data Streams and Operators. In *ICDT*, pp. 37-52, January 2005
- [29] B. Gedik, R. R. Bordawekar, and S. Y. Philip. CellJoin: a parallel stream join operator for the cell processor. *The VLDB journal*, 2009.
- [30] Apache Spark; <http://spark.apache.org/>. [Accessed Online 26/04/2017]
- [31] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proc. of SOSP*, pages 423–438, 2013.
- [32] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1345–1354, 1993.
- [33] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The hells-join: A heterogeneous stream join for extremely large windows. In *Proc. of DaMoN*, 2013.
- [34] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: reliable stream computation in the cloud. In *Proc. of EuroSys*, pages 1–14, 2013.
- [35] Diestel, Reinhard (2005), *Graph Theory* (3rd ed.), Springer, page 17.
- [36] Apache Kafka; <https://kafka.apache.org/>. [Accessed Online 26/04/2017]
- [37] Spring Cloud Stream; <https://cloud.spring.io/spring-cloud-stream/>. [Accessed Online 26/04/2017]
- [38] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of VLDB*, pp. 500–511, 2003.
- [39] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. Of VLDB*, pages 324–335, 2004.
- [40] Reactive Manifesto; <http://www.mammatustech.com/reactive-microservices>. [Accessed Online 26/04/2017]
- [41] RabbitMQ message broker, <https://www.rabbitmq.com/>. [Accessed Online 26/04/2017]
- [42] S. Vinoski, "[Advanced Message Queuing Protocol](#)" (PDF). *IEEE Internet Computing*. 10 (6): 87–89, 2006.
- [43] M. Logan, E. Merritt, and R. Carlsson, *Erlang and OTP in Action*, Manning Publications, November 2010.

- [44] O'Hara, J. (2007). ["Toward a commodity enterprise middleware"](#) (PDF). *ACM Queue*. 5 (4): 48–55.
- [45] AMQP, Protocol Specification; <https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf>. [Accessed Online 26/04/2017]
- [46] Win32 system message queues; ["About Messages and Message Queues"](#), *Microsoft Developer Network*. [Accessed Online 26/04/2017]
- [47] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, Semantics of Data Streams and Operators, *ACM Proceedings of the 10th international conference on Database Theory*, Jan 2005, pp. 37-52, Edinburgh, UK.
- [48] Spring Integration; <https://projects.spring.io/spring-integration/>. [Accessed Online 26/04/2017]
- [49] Openstack, Open-Source Cloud Computing Software; <https://www.openstack.org/>. [Accessed Online 26/04/2017]
- [50] Kubernetes Horizontal Pod Autoscaler; <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/> . [Accessed Online 26/04/2017]