NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMUNICATIONS

GRADUATE STUDIES PROGRAM
COMPUTER SYSTEMS TECHNOLOGY

MASTER THESIS

Elastic Infrastructure for Joining Stream Data

Nikolaos Petros Maravitsas

Supervisor: **Alex Delis,** NKUA Professor

ATHENS

July 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΕΧΝΟΛΟΓΙΑ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήση Ελαστικής Υποδομής για Σύζευξή Δεδομένων Ροών

Νικόλαος Πέτρου Μαραβίτσας

Επιβλέπων: **Αλέξης Δελής,** Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

Ιούλιος 2016

MASTER THESIS

Elastic Infrastructure for Joining Stream Data

Nikos P. Maravitsas
R.N.: M1284

SUPERVISOR: **Alex Delis,** NKUA Professor

EXAMINATION COMMITEE: **Mema Roussopoulos,** NKUA Associate Professor

July 2016

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Χρήση Ελαστικής Υποδομής για Σύζευξή Δεδομένων Ροών

Νίκος Π. Μαραβίτσας
Α.Μ.: Μ1284

ΕΠΙΒΛΕΠΩΝ: **Αλέξης Δελής,** Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Μέμα Ρουσσοπούλου,** Αναπληρωτής Καθηγητής

Ιούλιος 2016

ABSTRACT

In this work we aim to improve the performance of business intelligence applications, an important part of which is the Extraction-Transformation-Loading (ETL) processes. The vast majority of ETL processes involve very expensive joins between 'fresh' stream data flows and disk-stored relational data. We based our solution on an existing algorithm called Semi-Streamed Index Join algorithm (SSIJ), which successfully handles ETL transactions on a single computer node with very promising performance results. But we live in the era of information explosion. Large corporations have the ability to collect and store TBs of data every day. It is therefore necessary to move to a solution that uses multiple computing nodes. We developed an elastic distributed architecture that its main concern is the fair distribution of the computational load of SSIJ to multiple nodes. We have developed algorithms that efficiently direct the flow of the stream into clusters nodes in order to make caching as effective as possible. We also have the ability to add or remove compute nodes dynamically depending on the volume and speed of the stream traffic in order to maintain system performance stable and simultaneously avoid wasting valuable resources. In the implementation of this work we used containerized computing nodes which can operate in a cluster of virtual machines. We were based in Docker technology for containerizing our computing nodes. Our experiments were conducted in Google Cloud Platform. For the organization and scheduling of the Docker containers used the Kubernetes platform.

SUBJECT AREA: Stream Processing

KEYWORDS: stream processing, databases, big data, stream analytics, data warehousing.

ΠΕΡΙΛΗΨΗ

Σε αυτή την εργασία στοχεύουμε στη βελτίωση της απόδοσης των εργασιών επιχειρηματικής ευφυΐας σημαντικό κομμάτι των οποίων είναι οι εργασίες Εξόρυξη-Μετασχηματισμού-Φόρτωσης (ETL). Στην συντριπτική πλειοψηφία οι διαδικασίες ETL περιλαμβάνουν πολύ ακριβά joins μεταξύ δεδομένων ροών και σχεσιακών δεδομένων. Παρουσιάζουμε μια αρχιτεκτονική για την ελαστική προσαρμογή του αλγορίθμου Semi-Streamed Index Join (SSIJ) που με επιτυχία αντιμετωπίζει εργασίες τύπου-ETL σε ένανα υπολογιστικό κόμβο. Όμως ζούμε στην εποχή της έκρηξης των πληροφοριών. Οι μεγάλες εταιρίες έχουν τη δυνατότητα να συλλέγουν και να αποθηκεύουν TBs δεδομένων κάθε μέρα. Κατά συνέπεια είναι απαραίτητο να προχωρήσουμε σε μια λύση που χρησιμοποιεί πολλαπλούς υπολογιστικούς κόμβους. Αναπτύξαμε μια ελαστική κατανεμημένη αρχιτεκτονική που το βασικό της μέλημα είναι η δίκαιη διανομή του υπολογιστικού φόρτου του SSIJ σε πολλαπλούς κόμβους. Έχουμε αναπτύξει αλγόριθμους που κατευθύνουν αποδοτικά την ροή των δεδομένων μέσα συστάδες κόμβων, προκειμένου να κάνουμε αποτελεσματικό caching. Έχουμε επίσης τη δυνατότητα να προσθέσουμε ή να αφαιρέσουμε δυναμικά υπολογιστικούς κόμβους ανάλογα με τον όγκο και την ταχύτητα της κυκλοφορίας προκειμένου να διατηρηθεί η απόδοση του συστήματος σε σταθερά επίπεδα και ταυτόχρονα να μην σπαταλώνται πολύτιμοι πόροι. Στην υλοποίηση της εργασίας χρησιμοποιήσαμε containerized υπολογιστικούς κόμβους οι οποίοι μπορούν να λειτουργήσουν σε μια συστάδα απο virtual machines. Βασιστίκαμε στην τεχνολογία Docker για τους υπολογιστικούς μας κόμβους. Τα πειράματα πραγμοτοποιήθηκαν στην πλατφόρμα Google Cloud. Για την οργάνωση και την λειτουργία των Docker containers χρησιμοποιήσαμε την πλατφόρμα Kubernetes.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Επεξεργασία Ροών Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: επεξεργασία ροών δεδομένων, βάσεις δεδομένων, αποθήκες δεδομένων

Αφιερώνω αυτή την εργασία στους γονείς μου, Πέτρο και Βάσω, για την ακούραστη, αδιάκοπη και έμπρακτη υποστήριξή τους αλλά και για την εμπιστοσύνη που δείχνουν κάθε μέρα.

ΕΥΧΑΡΙΣΤΙΕΣ

Αρχικά θα ήθελα να ευχαριστήσω τον κ.Αλέξη Δελή που μου έδωσε την ευκαιρία να ασχοληθώ με αυτό το πολύ ενδιαφέρον θέμα. Επίσης τον ευχαριστώ για όλη την υποστήριξη καθ'όλη την διάρκεια της εργασίας αυτής αλλά και των σπουδών μου γενικότερα. Επιπλέον θα ήθελα να ευχαριστήσω τον συνάδελφο και φίλο μου Βαγγέλη Νομικό για την υποστήριξή του σε διάφορα τεχνικά θέματα της εργασίας. Τέλος, θα ήθελα να ευχαριστήσω θερμά τους κ.Σπύρο Σακελλαρίου και κ.Νίκο Λαουτάρη που χρηματοδότησαν την πρόσβαση μου στην πλατφόρμα Google Cloud όπου και είχα την ευκαιρία να πραγματοποιήσω τις απαραίτητες μετρήσεις και τα πειράματα για την ολοκλήρωση της εργασίας αυτής.

TABLE OF CONTENTS

TABLE OF FIGURES

FOREWORD

This project is the Final Thesis, with which my Master's Degree program in the National and Kapodistrian University of Athens is concluded.

1. INTRODUCTION

1.1 Business Intelligence

We are living in the area of information explosion. Large corporations have the potential of gathering and storing TBs of data every day. Consequently, along with data, there is an ever growing need for business automation that empowers organizations to better understand their data and make well informed strategic decisions and optimize the performance of operations.

Business intelligence technologies provide historical, current and predictive views of business operations. Common functions of business intelligence technologies are reporting, online analytical processing, analytics, data mining, process mining, complex event processing, business performance management, benchmarking, text mining, predictive analytics and prescriptive analytics.

Real time data analytics are among the most immensely growing paradigms when it comes to business intelligence these days. More and more corporations invest on active data warehouses, trying to cope with the information explosion.

1.2 Active Data Warehousing

As the ways of producing data are aggressively expanding, the sources of data are becoming more diverse. Data might be coming from within the organization, as the output of several business operations. There are always huge amounts of valuable historical data from legacy systems. Additionally, data is coming from other partner organizations, and finally from end users or customers. In a such a great variety of data sources, raw data need considerable cleansing, proactive transformations and filtering before actually getting stored in the warehouse and take part in the business analytics processes.

Extraction-Transformation-Loading (ETL) is still the most crucial part of these processes that perform this task traditionally during the refresh, off-line periods [1]. The refresh/offline periods include time periods during the day where the data warehouse is mostly inactive. The vast majority of ETL processes include very expensive joins between the fresh arrived records and some warehouse data or metadata tables. For example, record keys are often replaced with surrogates keys for compactness and consistency. This process, also known as conforming [1], necessitates the join of the refresh tuples from each source with a metadata table that relates keys and surrogate keys. Duplicate elimination or identification of newly inserted tuples provide more examples where similar join expressions are encountered [1].

The past few years there has been a considerable amount of work targeting ways to avoid the huge amounts of work that ETL had to perform with the normal workload of the warehouse. However, in emerging applications, such as network monitoring, supply-chain monitoring and sensory data analysis, as well as Internet of Things infrastructures, the latency introduced from the time that the data is entering the warehouse to the time it is ready for analysis may be unacceptably large. Even for traditional business intelligent tasks, finding the right piece of information at the right (i.e., shortest) time is a necessity for survival in today's competitive marketplace. Active data warehousing has emerged as a new BI paradigm where updates from the operational stores are propagated in (near) real-time to the repository.

This aforementioned shift of practices significantly affects the ETL process as the type of joins we described are now between an infinite stream of incoming records and some

stored data warehouse table. The output of this operation is a stream which typically participates in additional online operations.

As business analytics become more instrumental in strategic decision making, the Active Data Warehousing technology is maturing. This technology is engaged in integrating advanced decision support with day-to-day, even minute-to-minute decision making that increases quality.

Active data warehousing is an ever growing warehousing paradigm that supports real-time or near-real-time decision making. It is featured by event-driven actions triggered by a continuous stream of queries (generated by people or applications) against a broad set of relational disc-stored enterprise data. An active data warehouse presents an extension of the enterprise data warehousing capabilities. The analytical capabilities offered by this infrastructure are leveraged by responding to near-real-time business events as they occur, completing complex analyses upon demand, and alerting people or systems to take action.

1.3 Goals of the project

It is proven in practice that the most crucial components of such a system are:

a) a flexible/elastic infrastructure that supports the above, taking into account optimal resource acquisition depending on several performance and cost criteria.

b) dynamic deployment of resources and their orchestration

These are the basic goals of this project. We want to create a multi-node cluster with flexible and elastic characteristics where, depending on the stream traffic, it will add or remove nodes depending on the performance criteria that are set by the operator. This infrastructure will sit on an active data warehouse pipeline and will provide real-time information to the operators.

For this project we are going to take the basic ideas and key features of Antonios Deligiannakis et.al, in their paper "Semi-Streamed Index Join for Near-Real Time Execution of ETL Transformations". In this paper Deligannakis's team introduce a very high performance solution for joining stream data with a disc resident relation using a B+ tree as an indexing mechanism on the relation's key.

The main purpose of the SSIJ framework is to gracefully increase the performance of ETL processes. These processes mainly include joining a live stream of data with a relation stored in a hard drive. It incorporates indexes, an efficient dynamic memory allocation algorithm bases on the pattern of the stream as well as a innovative optimal read plan for reading the blocks of the relation on the disk. Using the above principals SSIJ managed to impressively outperform other semi-stream joining solution like MESHJOIN.

Although SSIJ has shown promising performance potential, its initial conception was to improve the performance of joining a stream with a relation near-real-time, on a single computing node. Its tested configuration was using a relation no bigger that 10GBs. And the various experiments that were conducted were alternating memory recourse allocation between 1% and 10% of the relation size, which translates for up to just 1GB of memory. And although this is relatively reasonable for a small data center will low volume needs and relatively low stream traffic, it just cannot cope in today's data explosion reality. SSIJ views semi-stream join operations from within the "single node" perspective.

The observations we have stated so far lead us to the following considerations. The most cost-efficient and performance effective way to tackle the above problem of hot-cold period alternations, is to have a system that elastically allocates and de-allocates its resources, according to specific dynamic criteria. Such criteria could include stream traffic volume, stored data volume, latency requirements and budget limitations.

To move to the directions of adaptive resource allocation, it is essential to abandon the "single node" point of view of SSIJ, and move to the multi-node point of view. In that sense the computational load of SSIJ should be distributed among many computing units. At the same time the computation needs to be distributed equally among the computational nodes, so as to make resource allocation much more efficient.

We developed a set of algorithms that let us achieve our goals. We created a multi-node flexible and elastic infrastructure that operate in the same way as [1] but in a distributed environment. We then deployed our implementation on Google Cloud Platform to perform experiments and see the results of our work.

2. THE SSIJ FRAMEWORK

2.1 SSIJ Introduction

In this chapter we are going to talk about the basic ideas and the key design features of the SSIJ Framework [1]. We are going to describe it's theoretical execution model as well as some of the insights behind its core algorithms. It is extremely important to get a good comprehension about this model, as our elastic framework is using it as its foundation, taking its basic ideas and incorporating aspects of modern distributed processing

2.2 SSIJ Purpose

The main purpose of the SSIJ framework is to increase the performance of ETL processes. These processes mainly include joining a live stream of data with a relation stored in a hard drive. It incorporates indexes, an efficient dynamic memory allocation algorithm bases on the pattern of the stream as well as a innovative optimal read plan for reading the blocks of the relation on the disk. Using the above principals SSIJ managed to impressively outperform other semi-stream joining solution like MESHJOIN.

2.3 SSIJ Basics

We start our SSIJ description by talking about its basic components. We are going to talk about its indexing infrastructure and its execution phases, the online phase and the joining phase. These two basic SSIJ characteristics are very important for our own execution model as we kept most of the ideas and basic thinking behind them intact. Some differences in those two components (the indexing and the two basic execution phases) will be mentioned in the next chapter.

Those two basic components are build on top of an efficient dynamic memory allocation strategy for buffering streaming data along with index blocks and relation blocks. Apart from the above, SSIJ's innovative idea relies on the incorporation of an optimal read plan for reading the relation blocks from the disk. Using the above principles, SSIJ managed to impressively outperform other semi-stream joining solution like MESHJOIN.

2.3.1 Index

The execution model of SSIJ makes use of an index on the joining attribute of the relation. It's basic requirement is to be able to retrieve a number of block ids where joining tuples of the relation reside on the disk. Many indexing solutions can be considered for such a task, but the choice was a B+ tree on the joining attribute, which is an adequate choice as most relational database management systems that are already optimized for joining operations, popularly create B+ indexes on the joining attribute.

In the SSIJ implementation the B+ tree index in use stores the relation tuples in the leaf of the tree. This is widely accepted model and has a the benefit of avoiding extra disk reads to actually read the relation block, as well as eliminate extra memory allocations and operations to find the correct tuple or tuples in the relation block. It and also decreases the indexing overhead as it reduces the index levels by one. Additionally they sate that for their execution process, they pin the upper levels of their B+ tree into the memory, but, potentially in a more resource conservative environment these index

blocks can be replaced with the usual cache replacement policy as with the pages containing the tuples of the relation. It is worth noting that an extra advantage of the B+ tree index is that normally the size of the non-leaf portion of the index is completely realistic and expected by most production environments willing to host relational database management systems. For instance the specific B+ tree index that was used for evaluating the performance of SSIJ was measured to have 13MB of inner node size for a 10GB relation [1].

2.3.2 SSIJ Infrastructure Components

In this section we are going to describe the basic structures that compose the SSIJ framework. The available memory for the systems is partitioned in five compartments. The size of these compartments is not fixed but dynamically allocated according to the needs of the system. These five compartments consists of the blocks of the aforementioned upper index levels, the cached blocks of the relation, two buffers regarding stream tuples and an inverted index. Here we are going to give some more detailed about each compartment and its usage.

1) **The index blocks:** This portion of the memory keeps pinned down the blocks of the upper lever of the B+ tree so as to be always available in memory. Using this memory buffer, no extra read operations are need for the indexing procedure and thus reduces the cost of index lookups, degrading it form disk to memory operations.
2) **Cached relation blocks:** This portion of the memory holds the blocks of the relation that have been read from the disk to in order to be joined with the stream. An important detail of this component is also a utility counter that is kept for each block to be used by the cache replacing policy.
3) **Input buffer:** This portion of the memory hold the stream tuples that arrive in the system and are waiting to be indexed and joined. This can be considered as the entry point of the system.
4) **Stream buffer:** For every stream tuple that is entering the systems, the first step is to index it, thus retrieve the ids of the relation blocks that need to be fetched in order to join the matching relation tuples with the stream tuple in question. There are two cases for those blocks. Either they are present in the cached relation blocks, or they require a disc read. All those stream tuples that require retrieving one or more blocks of the relation from the disc, we keep them in this special buffer called stream buffer. The required relation blocks will be later read with the aforementioned optimal read plan.
5) **Inverted Index:** For each relation block that needs to be read from disk (because some stream tuple in the stream buffers required its presence in memory), we maintain a list with the location of all matching stream tuples in SB for it. Multiple uses of the inverted index exist. Besides improving the performance of the join phase, the index is also important for efficiently guaranteeing the correctness of the overall process of SSIJ. The efficiency and effectiveness of cache replacements policy is also heavily based on this inverted index.

The next two images illustrate the various SSIJ components along with a table of their abbreviations. They are both taken from [1]. In the first image there is also a quick description of each components utilization. In the same image you can see in the red

numbers (from 1 to 6) the flow of the stream processing execution mechanism that SSIJ adopted.

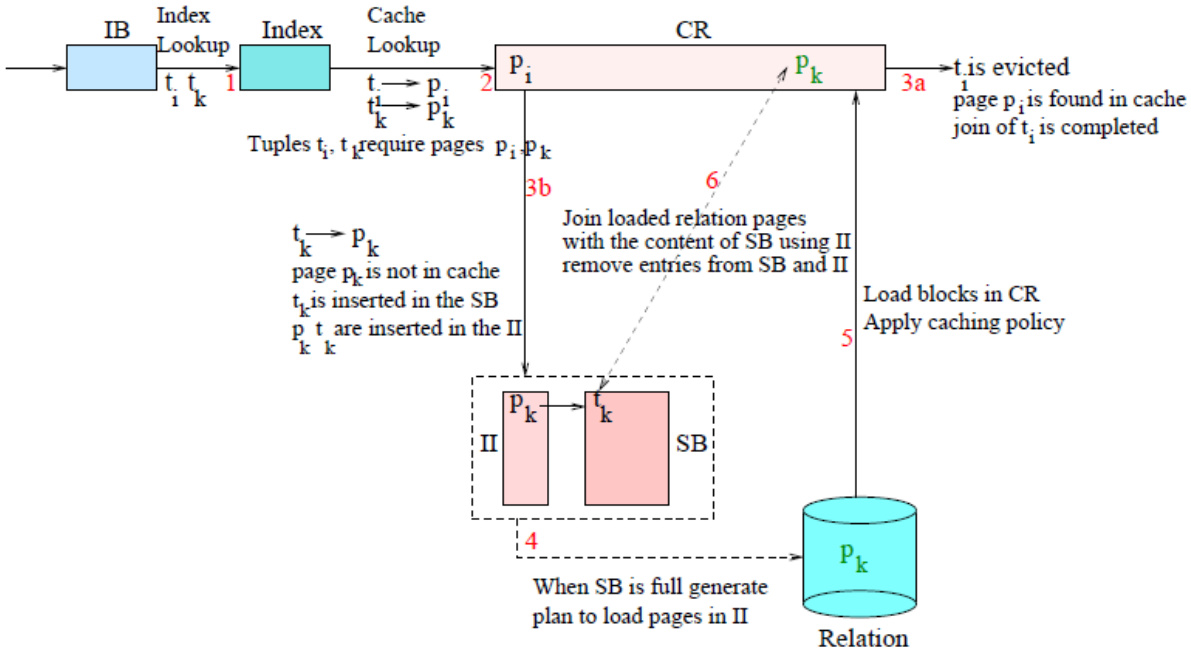

Figure 1: SSIJ Components and execution flow [1]

Symbol	Description
DR	Disk-resident relation
SR	Streaming relation (stream)
CR	Memory cached blocks/pages of DR
IB	Input buffer for unprocessed stream tuples
SB	Stream buffer: stream tuples waiting for disk blocks to be read for their join
IB_{thresh}	Minimum number of tuples in IB for the online phase to kick in
SB_{thresh}	Minimum number of tuples in SB for the join phase to kick in
p	A page of DR (cached or not)
s	A tuple of SR
II_p	Inverted index list for relation page p. Contains pointer to matching tuples in SB
M	Total memory allocated to SSIJ
P_u	Set of blocks that SSIJ must read in its *join phase*
S	Disk average seek time
T	Disk average transfer time of a disk page
$maxDist$	Maximum distance (in disk pages) of two pages b_1 and b_2 such that reading with one sequential I/O all blocks from b_1 to b_2 is cheaper than reading only b_1 and b_2 with two random I/Os
$maxScan$	Maximum length sequence of disk blocks that SSIJ may read with a single sequential scan

Table 1: SSIJ main symbols used [1]

2.4 SSIJ Stream processing Algorithm and Computational phases

In this section we are going to describe the basic ideas of the stream processing algorithm that SSIJ uses. We will talk about its three main phases of execution. During that process the utility and the very purpose of each of the components we described above will become clear. The next few sections are very vital for our own distributed execution model as we followed most of the principles described here.

2.4.1 Overview

Here follows a basic overview of the streaming algorithm. The execution flow is spited in three phases.

The first phase is called the ***Pending Phase***. During that phase the algorithms is waiting for the input buffer to accumulate a substantial amount of stream tuples before its main execution phase. There are lot of benefits to be extracted out of this techniques. First of all, it batches the execution of the next two phases. If the stream tuples where processed one by one the next phases of the algorithm would have to take place millions of times in a real system. Secondly, and most important, batching the stream tupes together allows SSIJ to take advantage of common access patters to the relation blocks that reside on the hard disc. It is vital for the performance of the algorithm to take advantage of those patterns in order to optimize the disc access when trying to fetch the relation pages to the main memory. Batching also makes memory management more efficient and effective as it gives a more clear and informed view of the relation blocks that are going to be needed in the online and join phase.

So after a solid amount of stream tuples has arrived in the input buffer, the execution enters the second phase which is called the ***Online Phase***. In the Online Phase every stream record from the Input buffer first goes through the *indexing process*. In the indexing process the joining attribute of the stream tuple is looked up in the index. At the end of this process, a list of relation block ids is returned that contain matching relation tuples. Immediately after the indexing process, the stream tuple is joined with every block from the previous list that is present in the cache. Of course some of the blocks from that list may not be cached, so we will need to retrieve the corresponding block from the hard disc. Every tuple that will trigger disc read is put in the stream buffer. When the stream buffer is full the ***Join Phase*** begins. In short, during the join phase, an efficient plan for reading the required blocks is calculated, the blocks are fetched from the disc to the main memory in the cached relation section and the join of the tuples is completed. In the next two sub-sections we are going to give some more details about the two major phases of the execution model, the Online Phase and the Join Phase

2.4.2 The Online Phase

When the online phase starts, the tuples that have arrived in the input buffer are sorted based on the characteristics of the index and the joining attribute. The sorted stream tuples will allow the indexing process that comes up to share scans of the index and of the cached relation blocks. That makes sense because a lot more that one tuple will follow the exact same path in the upper levels of the index as well as a lot of them will be joined with the same leaf nodes (in this case also relation blocks). So if we group together tuples in very close ranges with one another we will make a much more

efficient use of the cached relation and index blocks in memory, taking advange of locality. Next every tuple in the sorted sequence goes through the *indexing process*.

In the indexing process the joining attribute of the stream tuple is looked up in the index. At the end of this process, a list of relation block ids is returned. Each block in that list contains matching relation tuples. For all matching relation blocks that are in the cache, the join result is output immediately. The utility counter of any page in the cached relation memory portion is increased by one for every stream tuple is joined with. For each stream tuple s, if all the matching relation blocks for s are in the cache, then the join for s is complete, and s can be discarded.

If some matching relation blocks for s are not in the cache, the join with these blocks is not performed immediately. It it were to join the tuple with the non-cached blocks it would mean that the algorithm would pause until these blocks are fetched from disk. This is against the streaming nature of our processing model. Instead, the SSIJ algorithm will process the join of s with the disk resident matching blocks of the relation at a later point, during the join phase, in order to better cater for the cost of required read operations among several stream tuples. Since, at this step, we have identified the matching disk blocks for s, from the indexing process, we record this information by updating the inverted index, in order to speed up the join computation when these disk blocks are later retrieved from disk. The usefulness of the inverted index will become more evident in the description of the joining phase. Next , s is stored in the stream buffer SB . After the batch of stream tuples in the input buffer has been processed, the algorithm may move to the join phase.

2.4.3 The Join Phase

In the beginning of the join phase, we need to join all stream tuples accumulated in the stream buffer SB with their non-cached matching relation blocks. One of the most fundamental parts of this procedure for SSIJ is the calculation of a plan for reading the requested matching disk blocks. In more detail, we need to determine whether the required disk blocks will be read individually using random I/Os, or in larger sequences using sequential I/Os. This plan generation process requires that the disk blocks ids be sorted based on their physical layout; in the simplest case, this corresponds to sorting the offsets of the relation blocks on disk. This is performed in the first stage of the algorithm. It is important to note that the read plan may cause some disk pages to be loaded in spite of the fact that they are not requested by the stream, as part of a sequentially loaded disk segment that amortizes the I/O cost. Such “unwanted” disk blocks that are read because of sequential I/Os are evicted from cache immediately, since they have zero utility for the join.

During the join phase, the algorithm continuously reads sequences of disk blocks (based on the generated read plan). Each sequence of read relation blocks, as directed by the generated read plan, is fetched from disk and is inserted into the cache, replacing those cache pages with the lowest utility counters. The read disk blocks are then used to generate output join tuples by joining with the appropriate stream tuples, using the inverted index. After the join of the sequences is complete, the corresponding entries in the inverted index can be safely removed.

2.4.4 Cache Replacement policy

We are not going to expand on the Cache Replacement policies of SSJ as we did not incorporate them in our architecture. We decided to go for the classic LRU scheme at this version of our software. The reason for this is that there would be little performance gain in the context of the multi-node architecture where the true gains come from the increased memory resources.

A naive approach that maintains a sorted list of the page ids in the cache based on their utility counters, and on demand flushes the pages with the lowest utility counters, incurs a high cost. In particular, the repeated insertions in the list can introduce a large overhead when the amount of memory devoted to SSIJ is large. Moreover, during the online phase, the utility counters of the cached blocks are intensively updated, which requires continuous reorganization of the sorted list. Maintaining the cache blocks in a priority queue exhibits similar problems. The SSIJ implementation is based on the key observation that we do not need to actually read a page from the disc in order to calculate its utility counter, as we have all the necessary information in the inverted index of each page. Each entry in the II associates the id of a page that needs to be read from disk with the list of pointers to matching stream tuples in SB. Thus, the utility of a page in II is equal to the number of elements in the list it is associated with.

So, at the start of the join phase we know the utility counter of

1) all pages currently in the cache, and
2) all pages that will be read in the join phase.

At this point we have enough information in order to start making eviction decisions. More specifically, we sort the ids of the blocks in CR and the ids of matching disk pages (that need to be read) based on their utility. Considering the available memory (i.e., memory after subtracting the space needed for the input buffer, the stream tuples and the index structures), we determine the sorted subset, denoted **ToKeep**, of pages from pages that need to be read that should be in cache at the end of the join phase, given their utility. We also determine the sorted subset, denoted **ToRemove**, of block ids that are already in the CR and are going to be replaced by blocks in **ToKeep** with higher utility. If, during the join phase, the size of the cache is reduced due to the arrival of stream tuples, we can correspondingly increase the size of **ToRemove** or reduce the size of **ToKeep** according to the utility of pages in these lists. At this point we can simply evict any of the pages in **ToRemove**: these pages are not needed for the rest of the join phase and, given their utility, will not be in cache at the end of the join phase.

During the join phase, the algorithm continuously reads sequences of disk blocks (based on the generated read plan). Each sequence of read relation blocks, as directed by the generated read plan, is fetched from disk and is inserted into the cache, replacing those cache pages with the lowest utility counters. The read disk blocks are then used to generate output join tuples by joining with the appropriate stream tuples, using the inverted index. After the join of the sequences is complete, the corresponding entries in the inverted index can be safely removed.

We can potentially incorporate the above strategy in our system, because it is proven that it keeps the most useful blocks in the memory for a longer period of time. Additionally it evicts a number of blocks at once, leaving up more free space in a single operation.

3. DISTRIBUTED ELASTIC SSIJ ARCHITECTURE

3.1 Motivation

Although SSIJ has shown promising performance potential, its initial conception was to improve the performance of joining a stream with a relation near-real-time, on a single computing node. Its tested configuration was using a relation no bigger that 10GBs. And the various experiments that were conducted were alternating memory recourse allocation between 1% and 10% of the relation size, which translates for up to just 1GB of memory. And although this is relatively reasonable for a small data center will low volume needs and relatively low stream traffic, it just cannot cope in today's data explosion reality. SSIJ views semi-stream join operations from within the "single node" perspective.

Today's corporations accumulate terabytes of data every day. In this frantic pace, companies are constantly trying to provide more and more real-time applications on its customers and more business automation on its other departments. In such hectic environments ETL operations are becoming just a small part of a huge real time pipeline. So the latency requirements of such systems are constantly more strict.

Using the same analogies as SSIJ used for its experiments, what would be our memory needs when our relations is 100GBs, or 10TBs? As you can see it would be up to 1TB of memory. Despite the fact that multi-core single node enterprise machines with more 1TB of memory do really exist, the cost of such and investment is beyond the reach of most organizations that are trying to find their way in the Big Data era. And even if one company is big enough to have enough capital reserves and at the same time substantial volume of data as well as noticeable stream traffic that would justify this investments, the cost would be great in low traffic periods. Not many datacenters have constant traffic flow throughout the day, the week or the year. For a medium company which invested on that type of machine, low traffic periods lead to monetary loss.

Having said that, when we have high traffic periods. During these periods, the stream traffic reaches its peak and it requires more and more resources to be allocated in order to keep a constant or even increasing performance levels. The streaming nature and the low latency requirements of the problem, forbid the old manual resource allocation and de-allocation metrologies.

3.2 Assumptions and requirements

Seeing all the above one comes to certain conclusions. The most cost-efficient and performance effective way to tackle the above problem of hot-cold period alternations, is to have a system that elastically allocates and de-allocates its resources, according to specific dynamic criteria. Such criteria could include stream traffic volume, stored data volume, latency requirements and budget limitations.

To move to the directions of more resource allocation, it is essential to abandon the "single node" point of view of SSIJ, and move to the multi-node point of view. In that sense the computational load of SSIJ should be distributed among many computing units. At the same time the computation needs to be distributed equally among the computational nodes, so as to make resource allocation much more efficient.

3.3 Initial thoughts

In this section we are going to present the initial thinking behind the processes of transplanting all the SSIJ essential functionality into a distributed computational model. During the next paragraphs it will become vividly evident how SSIJ computational and execution model consists of all the important building blocks of a distributed system.

Don't forget that the purpose of the systems is the same as SSIJ's purpose. To join stream tuples with disc resident relation tuples. In that sense all the computing nodes of the system will have to be occupied during the whole process. While one computational node does one job, another node should does another job in order to fulfill the purpose of the system.

By a pleasant coincident, SSIJ seems to be almost destined for such a system. It already splits its computational workload into two main phases. The **Online Phase** and the **Join phase**. Additionally the several sub operations that are conducted during each phase are very distinct and most importantly, there are not confusingly depended to one another. The output of one operation, is the input of the next operation. The output of one phase is the input for the next phase.

3.4 Serial sub-functions of SSIJ

To be more specific, the purpose of the pending phase is to accumulate a substantial amount of stream tuples, so that the Online phase and the Join phase can be conducted in batches. So the output of the Online phase, which is an array of stream tupes, is the input to the online phase.

The online phase itself consists of one major and one minor sub operation. These two operations are conducted serially. Meaning that one of them takes the input of the online phase, does a computation, then passes an intermediate result to the next sub operation which finally does its own computation and produces the final result of the online phase.

The major operation of the online phase, which is also the first in order, is the indexing operation. We have described the indexing operation of SSIJ in great detail in section 2.4.2. All stream tuples have to go through the indexing process. For every stream tuple, the indexing process will produce a list of block ids of the disc resident relation, that have matching tuples that need to be joined. So after that computation, we can say each stream tuple is accompanied with the list of matching block ids.

That is exactly the input of the next sub operation of the online phase. The next sub operation of the online phase takes every tuple with its list of matching block ids and checks whether some of the blocks on that list are present in the cached portion of the relation. All the in memory blocks are immediately joined with the corresponding matching stream tuple and the result is flushed out. Every stream tuple that needs to be joined with a non cached relation block is passed to the next phase of the algorithm.

The next phase of the algorithm is the join phase. The join phase takes as input the list of tuples from the previous phase. Each tuple is accompanied by the list of blocks that were not cached at that moment. When enough of these tuples are accumulated, the join phase is responsible to fetch all the required blocks from the disc into the memory and join all stream records with the matching relation records and flush the result. This phase is responsible of computing a sensible plan to read the blocks from the disc. At the same times it need to take advantage all the knowledge it has about each block's utilization, so as to make informed decisions when cache replacement needs to take

place. A well-constructed cache replacement strategy is crucial for the performance of the system.

3.5 A pipeline execution pattern

The first task to do when trying to transform SSIJ from a single node algorithm to a fully fledged distributed system is think about how to distribute the computational effort among different computing nodes. As we saw in the previous section SSIJ's model fits perfectly to a distributed paradigm. That's because its individual sub-operations are quite distinct and are executed in a streaming/serial manner. In that sense, the output of one operation is the input of the next operation. That is very essential for our architecture.

This working framework perfectly suits a form of a **pipeline** execution pattern. The pipeline will consist of two major stages that correspond almost perfectly to SSIJ's execution phases. We are going to name the first stage the **Index Stage** and the second stage the **Join Stage**.

3.5.1 The Index stage

One can easily correlate the Online Phase of SSIJ with the Index Stage of our system. This first stage of our pipeline also includes the pending phase of SSIJ, where stream records are accumulated in the stream buffer. So our Indexing stage also acts as the system's input where stream records are accumulated so that they can be processed in batches by the next stages of our pipeline.

Every tuple that enters our system, needs to pass through the indexing process. Just as on the first step of the SSIJ's online phase, our Index stage is charged with the task of indexing every single stream tuple, and for every such tuple return a list of relation block ids that contain matching relation tuples that need to be joined. Recall that SSIJ's online phase also includes an actual join procedure. It joins the stream tuples with the portion of the matching relation locks that are already in the memory. We decided not to do that in the Index stage.

We dedicate this stage just for indexing. And that is an important decision because it completely decouples the two stages of our pipeline. It is crucial to achieve that because it leads to a more clean distribution of responsibilities for the participating nodes in the system. The indexing process itself is fairly simple and it also depends on the indexing structure that each system choose. We decided to follow SSIJ's decision to go with B+ trees. The index stage attaches to every stream tuple a list of matching relation blocks. This stream tuple accompanied by the list of matching relation block ids is the flushed to the next stage of our pipeline. The Join Stage.

3.5.2 The Join Stage

The Join Stage takes as input the result of the indexing stage. This is a list of stream tuples accompanied by the list of matching relation block ids. Every one of the relation block ids that is present in the cache, it is joined immediately with the corresponding tuple (in the same fashion as SSIJ's second sub operation of online phase). Every stream tuple that matches with a relation block that is not cached, and thus needed to be fetched from the disk to the memory, is pushed to a stream buffer cache. When the

stream buffer cache is full, the Join Stage has to fetch the required blocks from the disk to the memory and do the necessary join operations.

In this stage, we also utilize the inverted index, in a similar way as SSIJ. Its structures is of this form: We simply map every non cached relation block id to a list of matching stream stream tuples. You can see that structure in Figure 2.

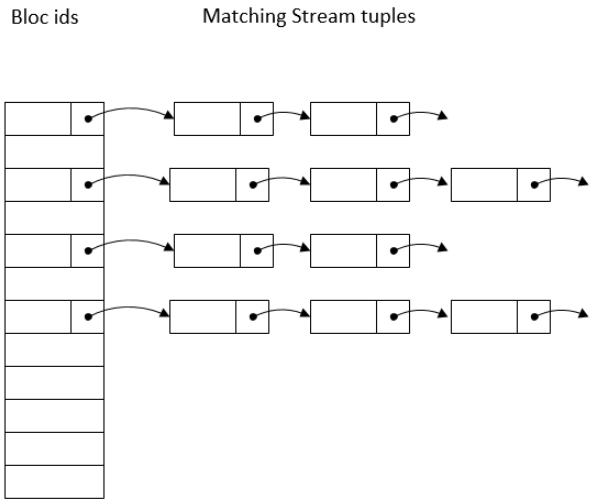

Figure 2: Inverted index

When number of the stream tuples in the above lists have reached a certain threshold, the Joiner stage moves to an operation similar to SSIJ's join phase. We read every relation block in the block id array and we join it with the corresponding list of stream tuples on the right. If there isn't sufficient space in the cache for that particular block, we need to replace one. We simply went for the fastest and well proven solution for that, a simple LRU implementation. That means that we simply remove the least recently used cached block.

Every block that is fetched from the disc, is pinned in memory until it is joined with all the tuples from its corresponding list on the right. Using this, we conform to the initial idea of a batched processing method for the stream, and at the same time, we make sure that each block is only read once from the disc during that phase. We don't risk removing a needed block that was just read.

It is important to note that the block ids in the array on the left (Figure 2) is kept sorted. This is done for an effort to read disc blocks in a serial manner as much as possible. We don't go as far as SSIJ goes to read blocks only with big serial reads (even if that means that it read unneeded blocks).

3.5.3 Task distribution

So far we have explained the pipeline patter that our distributed system would have. We have also decided that it is essential to distribute the tasks to multiple computing nodes. So the first natural approach would be to distribute the two pipeline stages among two computing nodes. The first node would perform the operations of the Index Stage we described above, and the second node would perform the operations needed for the Join Stage. These two nodes would be completely independent from one another and

would only communicate with each other via a network connection. This simplistic approach is illustrated in Figure 3.

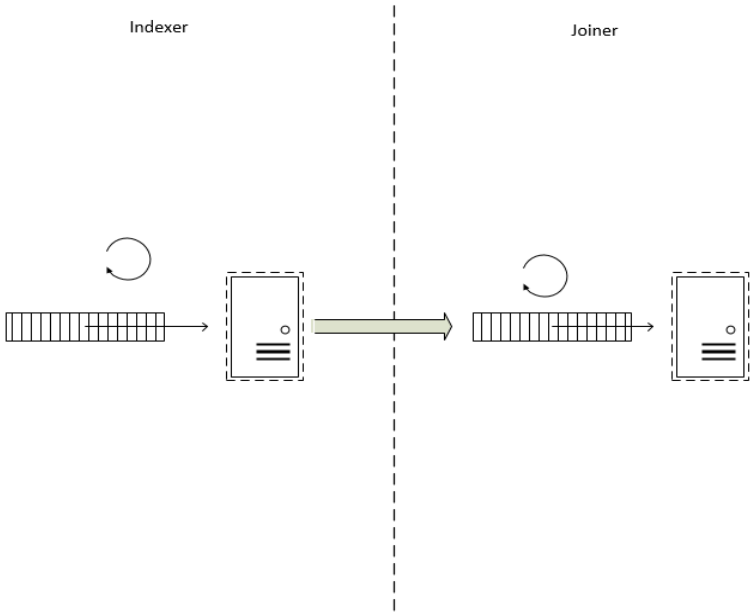

Figure 3: A possible first approach

As you can see, each node has its own input buffer. Of course, each node has its own memory resources. It is essential for the indexer node to have access to the hard drive where the B+ index resides. In the same manner, it is essential for the Joiner node to have access to the hard drive where the relation resides. It is important to mention that the two aforementioned drives could potentially be the same physical drive. But that is not at all necessary. The important aspect is that all the above units should be in very close proximity preferably in the same local network, so that network communication overhead is amortized by the increased computational resources and it is not actually a bottleneck for the whole system

As it is already evident from Figure 3, the first node constantly receives the stream from the external sources. After buffering a certain amount of them it starts the indexing procedure for every single one of them. When the result of the indexing for each stream tuple is produced, it is immediately flushed to the next node. The next node (that performs the join phase) can either choose to buffer the intermediate result and the enter its main process or it can actually start immediately even when a single indexed stream tuple has arrived. It is important for the next sections to also introduce a new notion here, that is the notion of an indexed stream tuple. An indexed stream tuple is the stream tuple itself along with the list of its matching relation blocks that have been computed from the index stage. We for our actual implementation we actually decided not to buffer again the indexed tuples, and start immediately the join stage upon indexed stream tuple arrival. This also conforms to the fact that we do not want to impose any delays for the joining of the stream tuples with an already cached relation block.

So the indexer constantly receives a stream of tuples and produces a stream of indexed tuples. The joiner then works on that stream of indexed tuples and it produces a stream of joined tuples. As joined tuple is a pair of two tuples. One stream tuple and one matching relation tuple. The pipeline nature of our system dictates that while the joiner performs joining computation, the indexer on the previous step constantly indexes

newly coming stream tuples, constantly feeding the joiner node. In the same sense, the joiner node constantly works on the indexed stream and produces the result of the whole system, a joined stream.

3.5.4 Multi-node architecture

What we have done so far is create a pipeline execution model for the SSIJ. We also created a distributed architecture that includes two nodes, each one of them executing the two stages of our pipeline. But this model, is very far from our initial goal that we have set for our system. Using just two nodes for instead of one is not a real change. We need to be able to allocate more resources for each stage of the pipepine.

A first approach would be to keep those two physical nodes, and just use multiple threads in each stage, instead of just one. Although this would increase the performance of our system, it is very questionable how good would it scale. Meaning that when the stream traffic increases and when the volume of my relation is large, how much would the system benefit from adding one more executing thread.

The answers is very little. And from a certain number of threads and above (a number bigger that the processing cores of that node) the performance would definitely drop because of costly context stitching and synchronization. But even if we don't take that cost into account, and we somehow created a completely lock-free algorithm, using very high performance concurrent data structures for the execution of our pipeline stages, after a certain point, it will not make any impact on the performance.

The reason is very simple and obvious. The main bottleneck of all systems that need to do extremely intense I/O operations is disc access. It is definitely not cpu performance. That makes sense as I/O operations still remain the most expensive computational unit, even with today's high performance SSD drives. And the only cure for that is caching. An I/O intensive system's performance is judged almost exclusively by its cache performance. That is why more and more high-end vendors invest on in-memory data grids. in-memory key-value stores, in-memory stream analytics. For example the most significant performance improvement over the popular Hadoop framework, was a release named Spark that used almost the same model, only used memory resources much more efficiently. Also in-memory databases are the latest trend for high performance real-time active datacenters.

We needed to stress out all the above in order to reveal the most valuable resource of our system, which should be obvious by now: it's memory. We need to be able to allocate more and more memory resources in order to provide linear scalability for our system and be able to hold the performance criteria we want independently of the volume of the stream traffic and the size of our relation.

As such, it becomes obvious that the only way to scale our system upwards allocating more memory, is to use multiple Indexer computing nodes and multiple Joiner computing nodes. A naive first approach on a multi-node architecture would be to have, let's say a number of indexers i and a number of joiners j. For example let's imagine that we have 4 indexers and 4 joiner nodes. Each node would have its own memory resources, its own cpu resources and so on.

The indexer nodes would all read from a common input buffer. Of course each indexer node would read different stream tuples. All indexer nodes would have access to the same volume that holds the B+ index. After indexing the stream tuples they are consuming, they would produce an indexed stream. All indexer nodes would output their

indexed stream to a common output buffer. This buffer would act as an input buffer for the next pipeline stage.

In the same manner, the Joiner nodes would all read from a common input buffer. Each node would read different indexed tuples. All joiner nodes would have access to the same volume that holds the relation. After joining the indexed tuples they are consuming, the would produce the output joined stream.

Such an architecture can be illustrated in Figure 4. As you can see indexers do not communicate with each other. Nor do the joiners communicate with each other. Similarly no indexer node communicates with joiner nodes and vise versa. This would also minimize the communication cost, which is a major concern when designing distributed systems. In a system like this the only communication cost is the input/output of the stream tuples for the indexers, and the indexed tuples for the joiners.

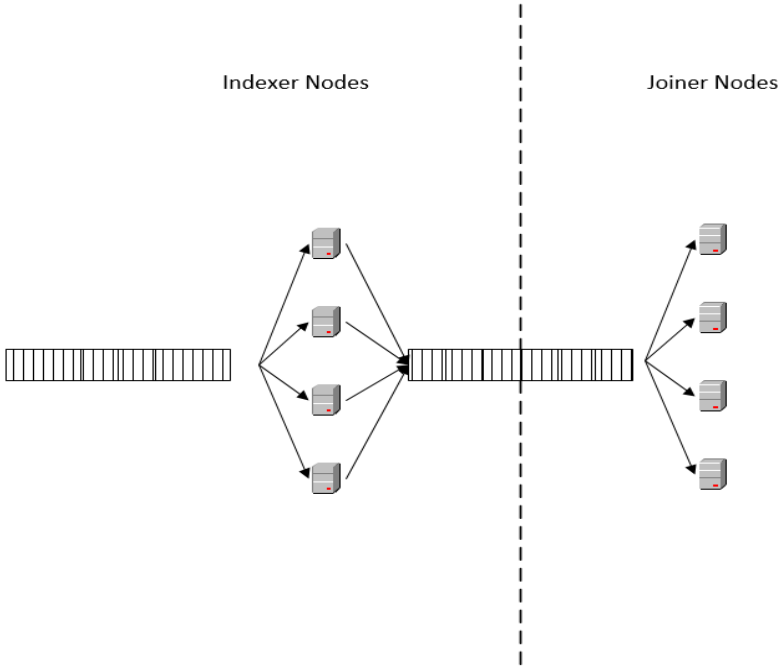

Figure 4: First multi-node approach

It is undeniable that if we allocate more computing machines this way, we will see a performance improvement. But when allocation more machines and investing in more resources one has to answer a number of very serious questions. How efficiently we are using those resources? Does our system scales linearly when allocating more? Are we using the available memory the best way possible? Is there any other configuration that would produce the same results with less resources? When designing a very high-performance, low-latency, massively scalable distributed systems, one has to be very considerate about the above questions.

For the architecture described by Figure 4, it is not hard to imagine that most of the above questions would fail. When indexers read randomly any stream tuple that comes in, in such non-orchestrated and unsupervised way, it is not guaranteed that we are using efficiently all the available memory on the indexers. And recall that, it is all about optimizing memory usage.

Let us consider a simple example. Imagine that we have 4 indexers, indexer0 to indexer3. We have a record tuple with an id x. In order to index that tuple we need to

read nodes A,B and C from the disc that contains the B+ tree, and cache those tree nodes into the memory. You can see that in Figure 5.

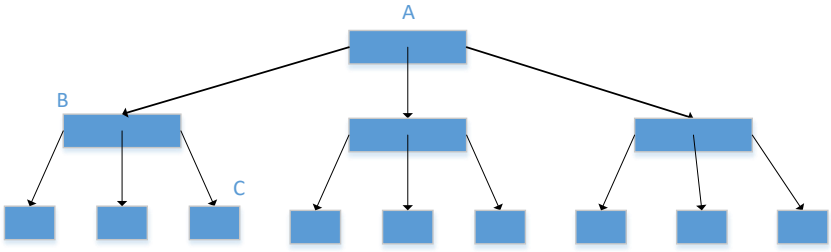

Figure 5: B+ tree div

Now imagine that this tuple is read by indexer0. So indexer0 will cache blocks A,B and C. Now the next stream tuple is y. This tuple also follows the same path as tuple x, meaning that we will need to read tree nodes A,B and C again. If we leave the input of the indexerS nodes non-orchestrated, probably an indexer other that indexer0 will read tuple y. But why should any other node other indexer0 be obligated to index typle y? indexer0 already has the required index blocks in its memory. If indexer1 has to index tuple y then he will need to also read blocks A,B and C from the disc. There is no point in doing that as indexer0 already has read and has cached those blocks.

The same principle also stands for the joiner nodes. For example if indexed tuples x and y require relation block R to be joined with, then the best plan is for those two tuples to be joined by the same joiner. If two different joiner cache the same relation block in their memory, then there is no point in allocation more computing nodes. We might as well allocate more threads for the computation, it is the same thing more or less, as we don't optimize the memory usage.

Thus the conclusion of the above discussion is that we need some units that will orchestrate the stream traffic through the indexer nodes and the indexed stream through the joiner nodes. These routing units will be responsible for two main tasks. The first and most important task is to route the stream traffic to specific computing nodes, so that we use all the available memory resources in the most efficient way possible.

The second task for these units is to scale the system up or down according to the performance requirement that the operator gives. For example when the stream traffic increases, or when it changes characteristics, we might need more memory to sustain a constant latency figure. On the other hand when the stream traffic is low we are probably wasting resources if we are way above the performance requirements. So we might need to remove resources from our cluster.

To achieve that we will add two more computing units that will be charged with the above two tasks. We will describe the new computing units in the next chapter, where we present the final architecture of our system.

3.6 A fully fledged elastic multi-node architecture

These two units that we are going to add, which will be responsible for shaping the stream traffic that each node will receive, have been called routers. The router responsible for distributing the input stream has been called **Index Router**, as it is the supervisor of the indexer nodes. The router responsible for the indexed stream has been called **Join Router**, as it is the supervisor of the joiner nodes.

The new flow of execution is illustrated in Figure 6. Now the input stream first passes through the Index Router. The Index Router keeps a catalog with all necessary information about indexer nodes. It then uses an algorithm to decide as quickly as possible to which indexer node each stream tuples should be forwarded to. Indexer nodes do not do anything different than before. The indexer nodes now produce the indexed stream that is directed to the Join Router.

The Join Router takes as input the indexed stream from the previous stage. The Join Router keeps a catalog with all necessary information about joiner nodes. It also uses an algorithm to decide as quickly as possible to which joiner node each indexed tuple should be forwarded to. The joiner nodes themselves do not do anything different than before. We have described in great detail the Indexer and Joiner node operations.

As you can see from Figure 6, all the communication in the cluster is one direction, it is forward only. There is no interleaved communication between computing nodes. That approach is necessary for scaling the computing nodes linearly.

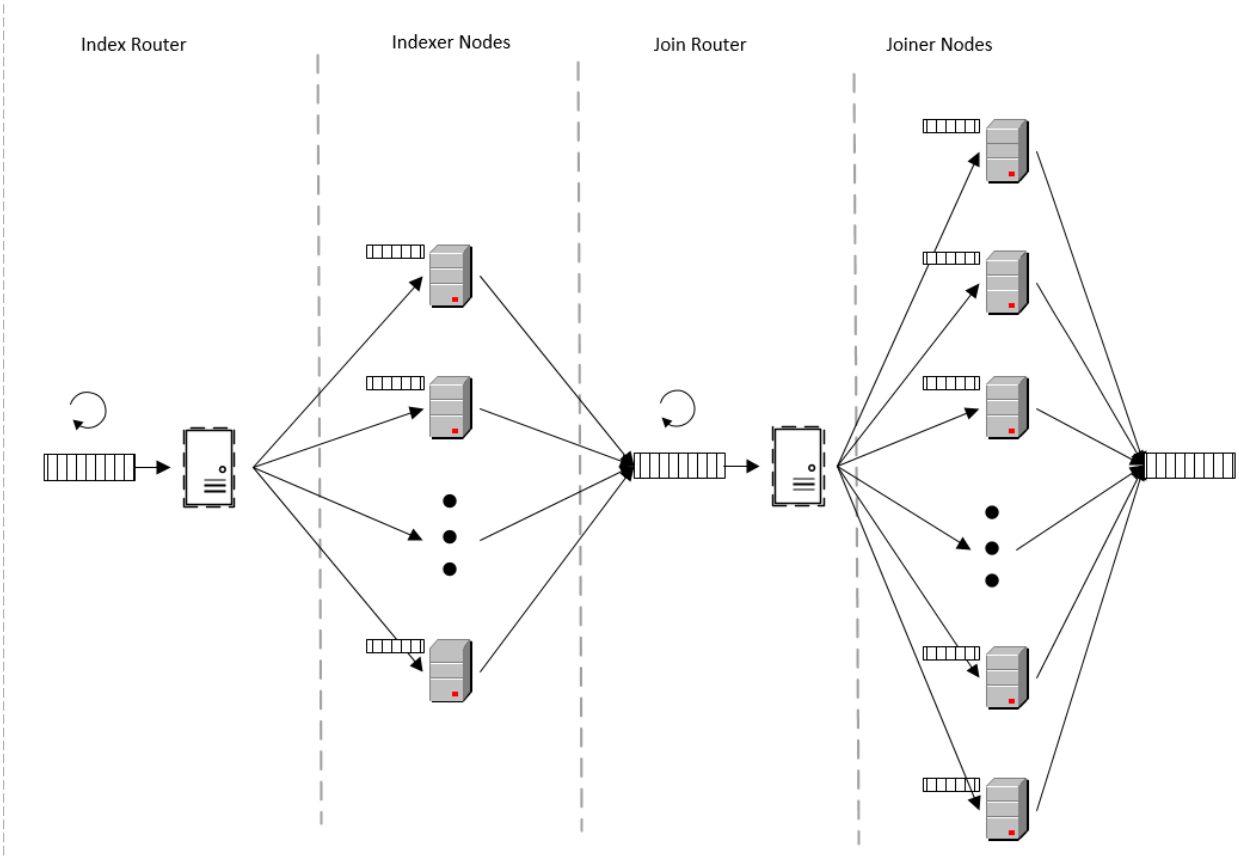

Figure 6: Distributed elastic SSIJ Architecture

The most interesting aspects of our systems is everything that happens inside the Index Router and the Join Router. This two nodes are responsible for the performance of the system.

3.7 The Index Router

In this section we are going to describe the basic operations of the Index Router node. The index router is responsible for taking the input raw stream of tuples and distribute every single tuple to one of the indexer nodes. There are several criteria to consider in order for index router to be fair amongst the indexer nodes.

1) First of all he has to distribute equal portions of stream tuples to every indexer over a certain amount of time. For example let's say that we have four indexer nodes and our index router has routed 10 million tuples after one minute, then each indexer should have to index 2.5 million tuples over that minute, approximately of course. It is natural that one indexer or the other would take more or less traffic but over a larger period of normalized traffic these skew should be amortized. So he should be, first of all, fair in the number of stream records he sends to each indexer node.
2) Remember that in our implementation, where we targeting huge relations, the B+ tree index is supposedly vast as well. So it is resident in a disc volume. Thus indexer node will have to do very heavy I/O. For every stream tuple they receive, they have to read as many nodes as is the depth of the tree. So the index router would have also to be fair in terms of the amount of I/O the makes every indexer to perform.
3) Another aspect that the index router should take into account is to use the memory resources of the indexers as efficiently as possible. This means that it should make the best effort to avoid duplicate entries in the indexer caches. To elaborate more on that image that if an indexer has cached B+ nodes A,B and C the index router should route to him the stream records that need A,B and C blocks to be indexed. An also it should avoid letting another node also cache A,B and C B+ nodes, because those blocks are already in the first indexer's cache. As we will see in the next few paragraphs, we cannot always achieve that non duplicate cache entries across the memory of all indexers.

3.7.1 Distribute the B+ tree search - Hot space distribution

In this section we are going to talk about the way that the indexer nodes distributes the B+ index search among the indexer nodes, in a way that it achieves as close as possible the three preconditions we talked about in the previous section.

The way the index router shapes the traffic of the incoming stream depends on three aspects:

1) The range of the index key.
2) The patter of the incoming traffic, meaning the range of the records keys that have arrived over a period of time.
3) The number of the available indexer nodes.

From the above, it is evident that we need to analyze the incoming stream tuples and perform analytic operations over their key values. In order to do that we are need to work over a substantial amount of tuples to come into safe conclusions over the characteristics of the stream.

To make the explanation of the whole process as easy as possible we are going to assume that the values of the tuple keys are arithmetic and the the distribution of the key values of the stream tuples is a normal distribution. The stream indexer works its analytics on the stream in a window manner. This window can be arbitrary big, but let's suppose it is 20,000 stream tuples. So over that 20,000 stream tuples, the index router calculates the maximum value, the minimum value and the mean value of the tuples keys. Let a be the min value, b be the max value and m be the mean value. We introduce the term **Hot Space** to characterize the space where the grand majority of key values falls into. In the example above the hot space is a sub space of [a,b]. It is not at

all necessary that in those 20,000 tuples the values are normally distributed. As the stream flows by the distribution will eventually normalize, but in a random 20,000 batch it is not always so.

In an effort to calculate a hot space as normally distributed as possible we do this. We calculate the distance m-a and b-m. We take the smallest result. If m-a is the smallest result it means that the majority of values falls in the space [a,m+(m-a)]. If b-m is the smallest result them the majority of values falls in the space [m-(b-m),b]. So one of these two spaces becomes the new hot space. Because it is possible that the boundaries of this host space, might be slightly different in every window, changing the hot space every 20,000 records might lead to great disturbances of the balance of the system. We need to maintain a constant hot space as long as possible. That's we why we are lazy in the hot space update. We update the hot space boundaries only if both of the current corresponding boundaries have shifted more than 10,000 units on either direction. To make thing more clear let's imagine that the current hot space is [a,b] and the hot space of the current 20,000 tuple window is [c,d]. If | a - c | and | b - d | are both bigger than 10,000 then the new hot space of the stream is [c,d]. We could potentially update each boundary individually but it was proven that this caused disturbances in the system.

Now that we have our hot space we need to distribute it across the indexers. Because the values in a hot space are normally distributed, we divide the hot space to subspaces equal to the number of indexer nodes. For instance, if we have 4 indexer nodes we are going to separate the hot space [a,b] into 4 equal subspaces: [a,a1), [a1,b2), [b2, b1) and [b1,b]. So now every tuple with key value that falls into the first sub space [a,a1) will be routed to indexer0, every tuple with key value that falls into the second subspace [a,b2]) will be routed to indexer1 and so on.

1) So these 4 indexer nodes are assigned 4 disjoint key spaces. This leads to a desirable result that covers our three preconditions

2) All indexers are going to be "fed" by the almost the same amount of tuples, as the tuples are normally distributed within the hot space.

3) Because each indexer node is assigned disjoint and equal spaces of keys, they will require more or less the same I/O operations to fetch B+ tree nodes from disc to memory space.

Because the subspaces are disjoint, each indexers is automatically assigned to search a different portion of the B+ tree. As the space of the stream tuple keys is expanding towards the space of the relation tuple keys, this is becoming even more accurate.

To provide a very simple explanation of how this algorithm works we are going to use a small example. Let's take for instance the B+ tree of Figure 7. An now let's suppose that we have 3 indexer nodes at this point. Also let's assume that the hot space at the moment is HS = [a,b], where a = <the smallest value of leaf node E> and b = <the biggest value of leaf node M>. This means that we have received stream tuples that cover the range of the relation keys. In that case, If we split the host space HS in three equal subspaces (as we have 3 indexers) the first subspace will include all values from leaf nodes E,F and G, the second subspace will include all values of H,I and J, the third subspace will include all values of K,L and M. So as you can see, as the index router is directing the stream tuples using those three subspaces of each indexer, the tree search is split among the three indexers nodes in a very balanced way.

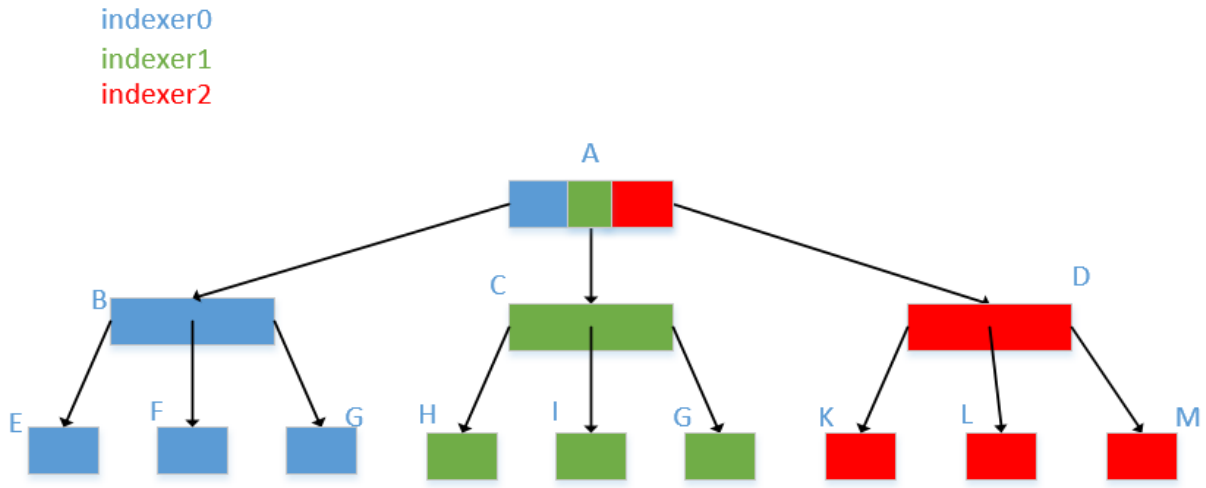

Figure 7: B+ tree search distribution

All indexers will cache node A, which is very natural as it is the root of the tree. But after that the set of blocks the three indexers need to read and cache is completely disjoint. Additionally they will need to read exactly the same amount of blocks, thus they will do the same I/O operations. But the most important thing is the efficient use of the total available memory. Using that technique, all the tree is cached among all indexer nodes, all indexer nodes have the same amount of blocks in their cache and additionally they have performed the same amount of I/O operations.

Of course the hot space is not always so balanced and the tree search is not always so perfectly distributed. For example if the host space only covered leaf nodes E to I, then the level of cached block replication would increase, meaning that indexers would have to cache the same tree nodes. This is not a big disadvantage. It has to do with the nature of the stream. If the stream has a very small range of keys, then we will have lot of replication for a stable given number if indexers. As the range of stream keys grows bigger and bigger, up to the point that it reaches the range of the relation keys, the search becomes more and more distributed.

3.7.2 Adding and removing Indexer nodes

So far we have seen how to distribute the traffic of the input stream towards a certain number of indexer nodes. In the introduction of this chapter we analyzed the need of being able to add or remove computing resources depending on the traffic the systems receives. So in this section we are going to talk about how the index router node adds or removes computing nodes depending on the traffic it is fed with, and also on the several performance aspect that an operator could give.

Let's imagine this scenario. At this point we have three indexer nodes. So the hot space is separated in three subspaces. Each indexer is assigned to index tuples that fall into one of the subspaces. Let's also assume that the performance characteristic that is the criterion of adding or removing computing nodes is cache hits. So let's assume that the minimum requirement for our indexer nodes is to achieve a cache hit percentage of 50%. Now let's say that the hot space is suddenly expanding and expanding. This means that a bigger portion of the B+ tree blocks are required for the indexing process. In order to keep a steady cache hit percentage, we now need to add more indexer nodes in order to increase available memory.

The way we do this is as follows: we have a utility counter for each of the subspaces. This utility counter measures the activity load of each of the spaces. While the hot space is expanding, some of the subspaces is receiving slightly bigger traffic that the others. We select the subspace with the higher utilization and split it into two subspaces. If more than one subspaces have similar utilization the we choose one randomly to split. We create a new indexer node and assign to it one of the new spaces.

If we still need to add more nodes, we repeat the same procedure. We always split the most used subspace. Because the newly create spaces are young, the spaces with the highest utilization will be one of the "old" subspaces. So, in practice, we keep splitting the "old" subspaces, until all are split. The same process goes on for the "new" subspaces when all the "old" spaces are split. Every time we split a subspace we assign the newly created space to a new indexer node.

This procedure will cause minimum disturbance to the system. When a space is split, its assigned indexer node will be responsible for less values. This means less B+ nodes to read. In turn, this will increase the cache hits immediately for this node. On the other hand the fresh indexer will need to fill its cache with new B+ blocks. Some of those nodes are surely available in the cache of another indexer, most probably a "neighbor" indexer ("neighbor" indexers are assigned "neighbor" subspaces). Potentially he could borrow those blocks from the other indexer or indexers, freeing some memory on his side as well. This would increase the complexity of system, so at this stage of the system we decided to leave the fresh indexer to read the blocks from the disc and "suffer" some more I/Os. We also let the old indexer remove unneeded blocks from its cache and replace them more valuable ones as fast as possible.

On the opposite direction we do the reverse of the above process when we need to remove an indexer. If, for instance in our example, when we added a node the cache hits reached 80% this means that we are probably wasting resources are the minimum requirement was 50%. So we need to remove an indexer node to cut costs. We do the exact reverse process. Now we choose the subspace with the least utilization and we merge it with one of its neighbors. If this subspace has more than one neighbors we choose the neighbor with the smaller utilization of the two. After removing one of the indexers we assign the enlarged subspace to its neighbor. Now this indexer has to cope with more values, meaning more b+ nodes, leading to decreased hit rates. We repeat the same process every time we want to remove an indexer. It is also important to note that during the above processes, any disturbances occurred in the system are only very localized around two neighbor indexer nodes.

3.8 The Join Router

In this section we are going to describe how the join router works. The index router is responsible for taking the stream of indexed tuples and distribute every single index tuple to one of the joiner nodes. Because there is no index here, the operations of join router are much simple that the ones of the index router.

Despite the fact that the join router process is much less complex, it tries to achieve the same basic goals as the index router. Let's briefly enlist them again:

1) First of all he has to distribute equal portions of indexed tuples to every joiner over a certain amount of time.

2) The join router would have also to be fair in terms of the amount of I/O he makes every joiner to perform.

3) Use all the available memory resources of the joiner as efficiently as possible. Try not to allow duplicate bocks to be cached by different joiners.

At this point, it is also worth reminding that the join router takes as input a stream of indexed tuples, that is, a stream of tuples that each one of them is accompanied by a list of matching blocks that it need to be joined with.

Now let's image the following scenario: we have 4 joiner nodes (joiner0 - joiner3). Stream tuple x needs to be joined with relation block i. So the corresponding indexed tuple is {x,i}. Now let's say tuple {x,i} is routed to joiner0. So joiner0 will eventually need to cache relation block i. Of course join router is aware of that. He knows that after tuple {x,i} is joined, then block i is surely in the cache of joiner0. It doesn't mean that block i will be in the cache forever, but definitely it will stay there some time, until it gets replaced by LRU. The amount of time block i will spend on the cache depends on the size of the cache buffer and the indexed records traffic. If we assume that a joiner node can hold up 5000 buffered relation blocks, then a newly cached block will definitely stay cached, for another 5000 ***distinct*** indexed tuples. By the term distinct indexed tuples we denote tuples that are joined with different relation blocks. Now after tuple {x,i}, the next indexed tuple in line is y that needs also to be joined with block i. Since block i is already in the cache of joiner0, it would be naive to redirect tuple {y,i} to an indexer other than i. Thus, the join router should direct this tuple also to indexer0, because that node already has block i cached.

It should be apparent what is the main strategy of join router, as it actually very simple. It keeps a map, correlating a relation block with the joiner node it was directed to. For instance after routing tuple {x,i} to joiner0, index router puts an entry in the map like so: *i -> joiner0*. Now when tuple {y,i} needs to be routed, the join router will look up its map and finds the previous entry. So he will direct {y,i} to joiner0 as well.

At first, this map is empty. So the choice of the joiner to send the first tuple is random. In fact let us assume that we choose in a serial manner. So the first joiner node to choose to direct an indexed tuple is joiner0. The join router is of course aware of the available memory that each node has. Let's assume that all joiners have 5000 buffer frames available, and that each buffer frame can hold a relation block. So each joiner node can have 5000 relation blocks cached.

The way the index router works is that it loads the cache of the joiners in a sequential manner. It first load up the memory of joiner0. That means that the first 5000 ***distinct*** indexed tuples will directed to joiner0. As you can imagine, it is very possible that joiner0 might initially receive way more than 5000 indexed tuples, as every tuple that comes that needs to be joined with the already cached blocks of joiner0 will be redirect to joiner0. When joiner0 has been give 5000 distinct indexed tuples, the join router assumes that his cache is full. If he sends another indexed tuple that needs to be joined with a relation block other than those 5000 distinct blocks, he knows that joiner0 will have to remove one of the blocks that he already has in his cache, will do an I/O to read the new block and cache it in the place of the replaced block.

But we do not need to do that as we have another 3 nodes with available memory. Additionally there is no escaping that I/O at this point as the new block is in none's cache. So we redirect that new stream tuple to joiner1. We repeat the same thing for joiner1 until his cache is full as well. In the mean time don't ignore the fact that every tuple that comes that needs to be joined with a block that is cached by joiner0, it is of course redirected to joiner0. That means, when eventually joiner1's cache is full, we have 10,000 distinct relation blocks in the cache, 5000 of those are in joiner0 and another 5000 of those are in joiner1. Up until now, no block has been replaced by another in the joiner's cache. We are still in the process of loading the whole available

memory. This process goes on until the cache of joiner2 and joiner3 are also full. At this point we have cached 20,000 relation blocks, 5000 in each joiner node.

Now if the next indexed tuple that comes requires to be joined with one of the 20,000 cached blocks, then it is routed to the joiner that has it. If it requires another relation block other than those 20,000, we are out of luck. We cannot escape evicting one of the cached blocks right now whichever joiner we choose to route the tuple to. In this case we select the victim 'node' serially again. So the first victim node is joiner0. As such, joiner0 will receive the tuple, evict the block that LRU dictates, load from the disc to the free frame the required block and do the join. In the meantime, the join router has added an entry to the map as we've explained before, in the form of *w -> joiner0*, where *w* is the aforementioned new block.

Now when index router redirected that tuple to joiner0, he was sure that joiner0 will need to replace one of its cached blocks, as he simply cannot hold more that 5000 distinct blocks. At first glance it seems beneficial for the join router to also know what block joiner would evict. Let that block be b, for instance. With that, he could remove the "old" entry *b->joiner0* from the map as it is no longer valid. Block b was evicted and it is no longer in the cache of joiner0. Potentially the join router could know what block will be evicted without having to communicate with joiner0. The joiner router knows exactly the sequence of block requests that went for joiner0. Since LRU is deterministic, meaning that the victim sequence will always be the same for a given block request sequence. So the join router could emulate the LRU algorithm for joiner0 and be sure what block would be evicted after his cache is full. But this would impose additional computational effort from the join router. An also we are considering the possibility of adding multi-thread execution on the joiner nodes. This would mean that each joiner will spawn multiple threads to perform his internal operations. At this scenario, it is not possible to predict the victim any more as the records are going to be consumed in a non deterministic way.

The important thing to note here is that it doesn't really matter, from a correctness perspective to know what the victim block would be. There will be no damage if we leave the invalid entry in the map. Imagine that block b gets evicted from joiner0. But the entry *b->joiner0* remains on the map. Now a tuple comes that needs to be joined with block b. Block b is in none's memory at this point. No matter what joiner we use he will need to evict another block from its cache and do an I/O for block b. So there is no harm routing it again back to joiner0, as the map dictates. Nothing different would happen if it was routed to any other joiner. None of the other joiners have it in their cache. To make that argument even stronger, since the index router doesn't know at this point what block will be evicted, if there is any possibility of block b being cached that it would be in the cache of joiner0.

So we are not compromising the correctness of the system with the aforementioned solution. But, it cannot go unnoticed that the map grows bigger and bigger. So after a certain limit, we do need to communicate with all the joiners in order to know the accurate state of their cache, meaning the exact list of relation blocks that they have in their caches. When the join router has that list in his hand, he can trim invalid map entries. But that only happens a few times over a big period of time. It also happens when we add or remove joiner node as you will see in the next section.

3.8.1 Adding and removing Joiner nodes

Similar to the indexer nodes, depending on the nature of the traffic, we might need to add more joiner nodes to keep with the performance standards that the operator of the system gives.

The process of adding a joiner node is fairly simple. When the new node is added it is treated as if an extra pile of memory was added to the system. Until he is also full, none of the already cached blocks should be evicted. We enter the same mode of operation as we did on the startup of the system, when we were loading the joiners caches. Thus now, every indexed tuple that turns up that is requiring a non - cached block, it is immediately routed to the new joiner node. This mode continues up until the new node's cache is also full. Afterwards the execution flows normally as we've described.

There is an important detail to consider about this operation. When the new node is added, we need to prevent all eviction from the old node caches. This means that, while the new node loads up its memory, we must ask any of the other joiner nodes to read blocks other than the ones they have in their cache. This means that have to accurately know each joiner's cache contents. If the danger is not yet evident let's consider again the previous example. Remember that we have kept and old entry *b->joiner0* in the map. Let's also suppose that block b is no longer actually in joiner0's cache. Now if we accidentally route an indexed tuple that requires block b to joiner0, he will need to evict another page. But we don't really need to do that because we now have available memory in the new joiner, thus there is no point evicting any cached block. That's why when adding a new node, just before starting routing traffic again, we communicate with the joiner nodes and update our map.

Removing a joiner node is quite simple. We simple remove all the entries from the map that contain that node and we stop routing traffic towards him. The operation then continues as it normally would if the gone joiner never existed.

3.9 Control Communication Channel

In Figure 6 we described the distributed architecture of our system and we showed the flow of communication between nodes. The bulk of communication between the nodes is accurately as depicted in Figure 6. It is one way with no strange interleaved interaction between nodes. Joiner nodes don't communicate with each other, joiner nodes don't communicate with each other. Despite that there still needs to be a backwards communication channel between the routers and the nodes they control. This channel will flow information from node to router. It will be called **control communication channel**.

We have described some of the reasons for node -> router communication in section 3.8 when we talked about the join router. Except the contents of the cache, the router nodes need to know more information about their worker nodes. The reason is that, routers will need to keep track of the performance characteristics of the system at any given time. That way they can decide whether to add or remove one of their worker nodes.

Thus there is a control channel that sends performance information from worker nodes back to their routers. It is worth noting that this communication happens in an frequency that depending on the speed of the stream (meaning how many stream tuples are received in the input per second). We can for instance assume that this communication happens every 20,000 records. If the stream speeds up, this communication will be

more frequent. If the stream slows down this communication will become less frequent. This skim makes sense as when the traffic is high we need to be up-to-date more quickly and on the other hand when the traffic is low we don't need frequent updates.

That way, the router nodes are well aware of the periods that the performance of the system is degrading and thus they need to allocate more resources and on the other hand the moments where the performance is way above the requirements so we need to remove resources to cut down costs.

4. IMPLEMENTATION

4.1 General information

In this chapter we are going to discuss several aspects that concern our implementation of the above distributed elastic SSIJ infrastructure. This implementation of Distributed SSIJ was done in Java. It consists of 12000 lines of code. This code includes the databases implementation, including the file manager, and of course the buffer manager which was used by all worker nodes (indexers and joiners). It also includes the implementation of the B+ tree index. Added to that, several data structure like buffers along with socket channels were also implemented in order to support the stream communication between nodes.

4.2 Relation file format and Index Choice

We chose the most generic format possible both for our relation files and our index files. Both heap files and data files are organized in blocks. Each block is 4096 bytes. That stands for both the relation files and the index files. For the data file each block is organized in heap form. Which means that the records don't have specific ordering in it. The relation file is organized in a heap format itself. Meaning that there is no specific ordering for the blocks themselves.

Having selected such a structure for our data file, we need to make certain decisions about the structure of our B+ tree index. First of all we used a primary index. That means that the index is upon the primary key of the relation. Now, the main concern was what would be the structure of the leaf nodes. We decided to go with the most generic implementation possible where we have the leaf nodes pointing to the corresponding data file blocks where each key is resides. And because we wanted to make the joining procedure as fast as possible, in the leaf nodes each key is accompanied with an number representing the offset of the record having that key in the data file block. To be more specific, if there are 240 records in a data block of the file relation each key in the leaf file is accompanied with a number from 0 to 240 denoting what is the relation record position in the block.

Thus these design options lead to decide to store the relation and the index in two different files. These two files can either reside on the same disc or they can reside on different physical volumes. The router nodes do not need to have access to either of those volumes. Indexer nodes need to have access to the volume where the index resides and joiner nodes need to have access to the volume where the relation resides.

4.3 Worker node granularity

As it is already evident the whole purpose of this project was to create a system that will allocate an de allocate working nodes on demand, in order to increase the memory resources of the system. The best possible scenario would be to be able to allocate physical dedicated computing nodes. A platform that supports that could be easily integrated in our system.

But today's businesses increasingly rely on Platform as a Service (PasS) Cloud frameworks provided by large corporations like, Google with Google Cloud, Microsoft with Azure, Amazon with Amazon Elastic Compute Cloud (Amazon EC2), IBM's SoftLayer which also provides infrastructure for Bare Metal physical machine allocation, RackSpace with its Dedicated Server infrastructures and many other smaller vendors.

4.3.1 Platform as a Service

Platform as a Service, often simply referred to as PaaS, is a category of cloud computing that provides a platform and environment to allow developers to build applications and services over the internet. PaaS services are hosted in the cloud and accessed by users simply via their web browser.

The infrastructure and applications are managed for customers and support is available. Services are constantly updated, with existing features upgraded and additional features added. PaaS providers can assist developers from the conception of their original ideas to the creation of applications, and through to testing and deployment. This is all achieved in a managed mechanism.

As with most cloud offerings, PaaS services are generally paid for on a subscription basis with clients ultimately paying just for what they use. Clients also benefit from the economies of scale that arise from the sharing of the underlying physical infrastructure between users, and that results in lower costs.

Using PasS services organizations don't have to invest in physical infrastructure. Being able to ‘rent’ **virtual infrastructure** has both cost benefits and practical benefits. They don’t need to purchase hardware themselves or employ the expertise to manage it. This leaves them free to focus on the development of applications. What’s more, clients will only need to rent the resources they need rather than invest in fixed, unused and therefore wasted capacity.

One of the most desired feature of PasS services is flexibility: customers can have control over the tools that are installed within their platforms and can create a platform that suits their specific requirements. They can ‘pick and choose’ the features they feel are necessary. Another one noteworthy feature is adaptability. Features can be changed if circumstances dictate that they should.

In summary, a PaaS offering supplies an operating environment for developing applications. In other words, it provides the architecture as well as the overall infrastructure to support application development. This includes networking, storage, software support and management services. It is therefore ideal for the development of new applications that are intended for the web as well as mobile devices and PCs.

Thus we decided to leverage the combination of 'read to go' configuration, flexibility and elasticity, along with the development friendly environment and use a PasS service for our cluster. So the maximum granularity of a worker node will be a virtual machine in a cloud service. We chose Google Cloud services because of some additional resources available on that platform what are of vital importance for our system.

4.3.2 Thread Workers

In the beginning of our development we decided to implement all the functionality of Distributed Elastic SSIJ using thread workers in order to emulate computing nodes. In that sense every thread worker was viewed as a computing node. It has its own memory resources, its own buffers and so on. Additionally the router nodes themselves were also threads. We've created some layers of abstraction in order to hide the node granularity. So router nodes are not aware whether their worker nodes are threads.

Because of the locality of the system, router nodes and worker nodes communicated with each other using local shared memory buffers. That was in an attempt to extract as much performance as possible by the 'locality factor'.

Although performance showed great potential, and the scalability was very close to linear (showing promises for the future), it started to degrade as more and more thread workers were allocated. This was due to the fact that despite the separation of memory areas in the threads, the whole system run on the same JVM. So the JVM memory management was becoming a deciding factor about performance. The bigger the memory needs, the more memory is allocated and so the bigger is the Garbage Collector overhead. Additionally the overhead of context switching between a big number of threads (think 100s of threads), was also evident. To make matters worse, we needed to use synchronization mechanisms for the access in the common stream buffers of the system. So synchronization overhead was also proving to be a serious bottleneck for the performance of the system.

We are aware that very low latency, extremely high performance, lock-free concurrent buffer data structures exists in various Java implementations. There are also well know techniques that avoid Garbage Collection altogether. This method have been used in the low-latency high-frequency trading markets for quite some time now, so as a result various Java libraries for that have come out. Most of these engineering attempts pre allocate all the necessary memory need, and use that memory pool the the needs of the system. For increased performance they use what is called direct memory. The hugely famous Java package ***sun.misc.Unsafe*** that provides manual native memory management and access is the backbone for most of those products and it is proven to have native performance (C/C++ levels of performance). Additionally some high-end optimized JVM implementations occur, with unnoticeable Garbage Collection overhead using truly concurrent Garbage Collection algorithms exist that provide a great JVM platform especially for low latency releases. One of the most famous is Azul's Zing JVM.

Although it could be really interesting and intriguing to see the performance results using a combination of the above technologies we leave of for future releases of the software as at this point we decided to focus on the algorithmic and architectural patterns of the system, and to provide a basic first implementation of the system.

4.3.3 JVM/Process workers - Socket Channel Communication

The next step of our development was to expand the granularity of worker nodes to processes. This was a vital step as those process would the actual processes that would run on the future where the granularity of the worker nodes would be Virtual Machines.

During this step we implemented also the necessary communication socket channel mechanisms. Now all the communication between workers and routers is socket-based. This allowed the further decoupling of the worker environment and the router's environment. To achieve high performance we used Java's NIO SocketChannel implementation. This socket implementation is optimized and widely deployed to support real-time streaming systems. We also implemented our own custom stream tuple serializers and deserializers that optimally serialize a stream tuple to an array of bytes and vise versa. We did that in order to avoid costly Java-native serialization methods, even from that first release.

In our implementations all nodes that receive input open a Server Socket. All nodes that want to send data to those nodes need to create a client socket connection with the remove server socket. If a node wants to both receive and send data he will do both of the aforementioned actions. All the socket streams are none blocking. We use the Java provided poll/select mechanisms that provides us with information about what channels

are available for reading and writing. It worth noting that this selection mechanism uses the native poll/selection mechanism that the underlying operating system provides.

In our implementation we decided to dedicate a thread on the input and deserialization operation. All nodes have their own input memory buffers. The input thread is responsible for reading the stream from the socket, deserializing it and then providing it to the main thread where all the operations take place. Every processes spawns a separate thread to support input operations.

Now there are no common buffers used and no common memory used between computing nodes. So synchronization cost is out of the way (except for the input thread and the main thread in each process). Now every node runs on its own JVM so Garbage Collection cost is also lower as well. And in fact any Garbage Collection cost will now have only node-local performance effects.

At this point we had also to think about the implementation of the control channel. We could use the same technologies, meaning Socket channels and also create our own serializes for the required messages, but for this specific task we decided to use a Java provided solution. We used Remote Method Invocation protocol for that matter. We created all the necessary objects and interfaces to support it. It is worth noting that RMI is also a socket based protocol. The Java Remote Method Invocation (Java RMI) is a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage collection. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP). There are also more high-performance message passing protocol implementation for Java, like ActiveMQ and RabbitMQ. At this version of our software we decided to use RMI.

In short, the way we use RMI is that every worker node opens an RMI socket and exposes a certain API to the RMI server that runs on his JVM. Then, the router nodes connect to that remote RMI socket and invoke remote methods to find out the worker-local information they need to know.

4.3.4 Container workers

In this section we are going to talk about containers and why we wanted to containarize our worker nodes. The reason we wanted to containerize our worker nodes is very simple. Most PasS platform offer Managed Container Cluster Engines that help you containerize your applications and deploy them on a local cluster that also has the capability of autoscaling (we will talk more on that later). You can use specific apis to add or remove containers on demand. That seems to fit perfectly our model of execution so we decided to look more into container-based virtualization.

Container-based virtualization, also called operating system virtualization, is an approach to virtualization in which the virtualization layer runs as an application within the operating system (OS). In this approach, the operating system's kernel runs on the hardware node with several isolated guest virtual machines (VMs) installed on top of it. The isolated guests are called containers.

With container-based virtualization, there isn't the overhead associated with having each guest run a completely installed operating system. This approach can also improve performance because there is just one operating system taking care of

hardware calls. A disadvantage of container-based virtualization, however, is that each guest must use the same operating system the host uses.

Operating-system-level virtualization is commonly used in virtual hosting environments, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users. System administrators may also use it, to a lesser extent, for consolidating server hardware by moving services on separate hosts into containers on the one server.

Other typical scenarios include separating several applications to separate containers for improved security, hardware independence, and added resource management features. Another strong case to also use containers is that Operating-system-level virtualization implementations capable of live migration can also be used for dynamic load balancing of containers between nodes in a cluster.

Additionally, as we've already mentioned Operating-system-level virtualization usually imposes little to no overhead, because programs in virtual partitions use the operating system's normal system call interface and do not need to be subjected to emulation or be run in an intermediate virtual machine, as is the case with whole-system virtualizers (such as VMware ESXi, QEMU or Hyper-V) and paravirtualizers (such as Xen or UML). This form of virtualization also does not require support in hardware to perform efficiently.

4.3.5 Docker

Docker is undoubtedly the most famous and widely used Linux Container implementation. Docker [7] [8] allows you to package an application with all of its dependencies into a standardized unit for software development. Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in.

Docker containers spin up and down in seconds making it easy to scale an application service at any time to satisfy peak customer demand, then just as easily spin down those containers to only use the resources you need when you need it.

A Docker image is made up of filesystems layered over each other. At the base is a boot filesystem, bootfs, which resembles the typical Linux/Unix boot filesystem. A Docker user will probably never interact with the boot filesystem. Indeed, when a container has booted, it is moved into memory, and the boot filesystem is unmounted to free up the RAM used by the initrd disk image. So far this looks pretty much like a typical Linux virtualization stack. Indeed, Docker next layers a root filesystem, rootfs, on top of the boot filesystem. This rootfs can be one or more operating systems (e.g., a Debian or Ubuntu filesystem).

In a more traditional Linux boot, the root filesystem is mounted read-only and then switched to read-write after boot and an integrity check is conducted. In the Docker world, however, the root filesystem stays in read-only mode, and Docker takes advantage of a union mount to add more read-only filesystems onto the root filesystem. A union mount is a mount that allows several filesystems to be mounted at one time but appear to be one filesystem. The union mount overlays the filesystems on top of one another so that the resulting filesystem may contain files and subdirectories from any or all of the underlying filesystems. Docker calls each of these filesystems images. Images

can be layered on top of one another. The image below is called the parent image and you can traverse each layer until you reach the bottom of the image stack where the final image is called the base image.

Finally, when a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below. This is where whatever processes we want our Docker container to run will execute.

In our container cluster we are going to use two different Docker Images. One Docker for the Indexer nodes and one Docker image for the Joiner nodes. We decided that the router nodes will run on dedicated VMs in the underlying VM cluster. Those two docker images will be used to create two different kinds of containers. Indexer Containers and Joiner Containers. Each one the Joiner containers will run the Joiner JVM/process and each one of the Indexer Containers will run the Indexer JVM/process. So basically each container will just run a JVM with the corresponding process.

4.3.6 Kubernetes Framework

The Kubernetes [9] project was started by Google in 2014. The name Kubernetes originates from Greek, meaning “helmsman” or “pilot”, and is the root of “governor” and “cybernetic”. K8s is an abbreviation derived by replacing the 8 letters “ubernete” with 8.

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure. With Kubernetes, we are able to quickly and efficiently respond to user demand. We can deploy our applications quickly and predictably. Kubernetes allows us to scale our applications on the fly. We can optimize the use of our hardware by using only the resources we need. Kubernetes goal is to foster an ecosystem of components and tools that relieve the burden of running applications in public and private clouds. Kubernetes is portable: public, private, hybrid, multi-cloud extensible: modular, pluggable, hookable, composable self-healing: auto-placement, auto-restart, auto-replication, auto-scaling.

The Old Way to deploy applications was to install the applications on a host using the operating system package manager. This had the disadvantage of entangling the applications’ executables, configuration, libraries, and lifecycles with each other and with the host OS. One could build immutable virtual-machine images in order to achieve predictable rollouts and rollbacks, but VMs are heavyweight and non-portable.

The New Way is to deploy containers based on operating-system-level virtualization rather than hardware virtualization. These containers are isolated from each other and from the host: they have their own filesystems, they can’t see each others’ processes, and their computational resource usage can be bounded. They are easier to build than VMs, and because they are decoupled from the underlying infrastructure and from the host filesystem, they are portable across clouds and OS distributions.

Because containers are small and fast, one application can be packed in each container image. This one-to-one application-to-image relationship unlocks the full benefits of containers. With containers, immutable container images can be created at build/release time rather than deployment time, since each application doesn’t need to be composed with the rest of the application stack, nor married to the production infrastructure environment. Generating container images at build/release time enables a consistent environment to be carried from development into production. Similarly, containers are vastly more transparent than VMs, which facilitates monitoring and

management. This is especially true when the containers' process lifecycles are managed by the infrastructure rather than hidden by a process supervisor inside the container. Finally, with a single application per container, managing the containers becomes tantamount to managing deployment of the application.

At a minimum, Kubernetes can schedule and run application containers on clusters of physical or virtual machines. However, Kubernetes also allows developers to 'cut the cord' to physical and virtual machines, moving from a host-centric infrastructure to a container-centric infrastructure, which provides the full advantages and benefits inherent to containers. Kubernetes provides the infrastructure to build a truly container-centric development environment.

Kubernetes satisfies a number of common needs of applications running in production, such as:

- co-locating helper processes,
- facilitating composite applications and preserving the one-application-per-container model,
- mounting storage systems,
- distributing secrets,
- application health checking,
- replicating application instances,
- horizontal auto-scaling,
- naming and discovery,
- load balancing,
- rolling updates,
- resource monitoring,
- log access and ingestion,
- support for introspection and debugging, and
- identity and authorization.

This provides the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and facilitates portability across infrastructure providers.

This was the next step in our implementation. As you might remember the final goal of the system was to deploy the worker nodes in a Virtual Machine cluster. We also wanted the ability to add more virtual machines when the performance of our system degrades and remove virtual machines when the performance of our system overcomes the required standards and we want to save costs.

4.3.7 Google Cloud Container Engine

Google Container Engine [10] is a powerful cluster manager and orchestration system for running our Docker containers. Container Engine schedules our containers into the cluster and manages them automatically based on requirements the operator defines

(such as CPU and memory). It's built on the open source Kubernetes system, giving us the flexibility to take advantage of on-premises, hybrid, or public cloud infrastructure.

One can set up a managed container cluster of virtual machines, ready for deployment in just minutes. The cluster is equipped with capabilities, such as logging and container health checking, to make application management easier.

You can also declare your containers' requirements, such as the amount of CPU/memory to reserve, number of replicas, and keepalive policy, in a simple JSON config file. Container Engine will schedule our containers as declared, and actively manage your application to ensure requirements are met.

With Red Hat, Microsoft, IBM, Mirantis OpenStack, and VMware (and the list keeps growing) working to integrate Kubernetes into their platforms, we will be able to move workloads, or take advantage of multiple cloud providers, more easily.

Google cluster infrastructure offers a very flexible auto scaling feature that helps optimize resource efficiency. When you want to deploy your application on the cluster you have to set an initial cluster size. If that cluster size is let's say 5, then 5 VMs are going to be created and they are going to be dedicated for your container cluster. You can then specify several performance standards that you want your cluster to meet. For example you can set a CPU threshold. If the CPU usage of the cluster is more than that threshold, then more Virtual Machined are automatically added to your cluster. On the same spirit when the CPU usage greatly overcomes that threshold the engine will remove some of the virtual machines

In our case because in the configuration we wanted our containers to take all the available resources at each machine, that auto scaling feature worked well, because more machines where dynamically added on the cluster when there was need. Now using this feature we can eventually meet out initial goal. If we set up our containers to use all available VM resources and at the same time set the performance thresholds of the cluster relatively high then each container will be run on a different VM which is the initial design.

But this is not always necessary. For example when our traffic is low and the performance is quite sufficient at that point, there is no need to have all containers run on different VMs. They could run on the same VM if the performance standards are met. It is up to VM and container cluster management platform to decide whether to add more VMs in the system. We have benefited greatly from this because Google Cloud platform already takes into account our cost conservative perspective and only scales up the VMs when it is necessary, helping the user save costs.

5. DEPLOYMENT, EXPERIMENTS AND RESULTS

5.1 Google Cloud Deployment and Setup

In order to conduct our experiments we create a master Virtual Machine in the Google Cloud Compute Engine service. This was a machine with 64bit 4Core 2.6GHz with 14GB or RAM and 50 GB HDD. We decided to use this machine as the resident computing node for our two router nodes. The worker nodes will run on separate containers. In order to achieve our initial goal "one container per vm" we created an initial cluster of 20 VMs. Each of these machines has a 2Core 2.6 GHz CPU with 6GB of RAM.

Now in order to deploy our containers into the VMs we used Kubernetes as our "cluster master". For each one computing node we create a new Kubernetes Pod. A pod is a group of containers that are scheduled onto the same host. Pods serve as units of scheduling, deployment, and horizontal scaling/replication. Pods share fate, and share some resources, such as storage volumes and IP addresses. Now in our case we will use single container pods. A single container pod has only on container running it. We did that because we wanted to be able to run each container on a separate VM.

One major problem with the current version of Kubernetes is that no more than one pods can be attached to the same external volume. This meant that we had to have a local copy of the index file in every indexer and a local copy of the relation file in every joiner. This is a major complaint for the Kubernetes platform, and as a very large portion of the community has been disappointed by such a decision, Kubernetes Development team is planning to add that feature in the next Kubernetes release. Granted, Kubernetes is an extremely young project, considering its granularity and its very ambitious goals.

Kubernetes offers a GUI manager interface that helps you very quickly create pods and deploy applications. But most importantly it offers a fully fledged REST API that enables programmatic access to its full functionality. We've found a newly created Java library that wraps around that REST api and enables client code to fully leverage the power of Kubernetes. That library is called Kubernetes-Client and it is created by the Fabric8 development team. Fabric8 is an orchestration framework that sits above Kubernetes and Docket, and it supervises the creation of pods and deployment of applications.

Using that API we are able to create pods and thus containers on demand. There is also another important Kubernetes primitive that enables you to create resilient pods. It is called Replication controller. In the replication controller you can specify the number of replicas you want to create for a pod. If you specify that you need 4 replicas for example, Kubernetes makes sure that 4 replicas of your pod are always running . This is a very useful feature that helps you create resilient multinode systems.

But we wanted to create an destroy our containers on demand. We assume that the operator of our own system will put in several performance criteria that we need to meet and accordingly our system should add or remove containers. We decided to use one performance criterion that has to do with the most valuable asset of our cluster, memory. So we decide that to be cache hit ratio. Cache hit ratio is a vital and deciding factor about the whole systems performance. So we give the two routers a specific hit ratio that should be the target total hit ratio for their corresponding worker nodes.

The two routers would have to create or remove computing nodes in order to conform to that specified number. The velocity with which the routers add or remove computing nodes have to do with what is the current and the target hit ratio. If there is a lot of distance between the current hit ratio and the target hit ratio the routers aggressively

add more nodes in order to catch the desired hit ratio as quick as possible. On the same spirit, if the current hit ratio is way above the desired target ration then we remove computing nodes to get as close as possible to the target. In order not to get trapped between continuously adding and removing computing nodes we give a margin of +-0.1 to the target hit ratio. This means that if we are 0.1 away from the target, either over or under, we don't add or delete any nodes.

For our experiments we create a relation with 300,000 blocks containing 153,299,999 records with a size of approximately 2GB. Our system is indented to be used with much larger volumes of data, but in this first prototype version we settled for a smaller database to mainly focus on testing the efficiency of resource allocation, e.g scalability and elasticity. The records have an integer key. Keys are all multiples of 5 ranging from 0 to 766,499,995. The B+ index tree has 3 levels and a total of 453,986 blocks. That is an extra 153,986 blocks of internal nodes added to the leaf level of 300,00 blocks.

In order to create our worker pods we have created two different Docker images. One Docker image that runs the joiner process and a second Docker image that runs the indexer process. Those two Docker images were uploaded in the Google Container Registry. This is a Docker Image Registry that is resident on the Google Cloud. Then you can easily create your pods just specifying the name of the image you want your container to start (along with other configuration parameters of course).

For our tests, every indexer has a buffer of 50,000 blocks. Every joiner has 5000 blocks of buffer capacity. .. .Kubernetes offers a GUI manager interface that helps you very quickly create pods and deploy applications. But most importantly it offers a fully fledged REST API that enables programmatic access to its full functionality. We've found a newly created Java library that wraps around that REST api and enables client code to fully leverage the power of Kubernetes. That library is called kubernetes-client and it is created by the Fabric8 development team. Fabric8 is an orchestration framework that sits above Kubernetes and Docker, and it supervises the creation of pods and deployment of applications.

The stream we are going to use is going to be created by a generator. The generator will run on the same VM as the index router but he is communicating with him via socket and not using share memory buffers. The key of the stream tuples is going to be an integer. They are going to be distributed in Gaussian form. The range of the keys is going to vary from experiment to experiment. We are going to perform experiments where the key range is from 0 to 300,000,000, covering 50% of the relation tuple range, and from 0 to 766,499,995 covering 100% the range of the relation keys. Most of the experiments are going to be run for a period of 4 to 6 minutes.

5.2 Scalability measurements

There are several aspects to consider when measuring the scalability of such systems. And that is the fact that because there are several stages in the pipeline, having a bottleneck in one can seriously affect the performance of the other. So for the measurements that follow we are going to have two different kinds of tests. On the first test we are going to measure the throughput of the joiners (and that is also the throughput of the whole system). but we are going to allocate as much resources as possible for the indexer nodes so as not to let the index stage be a bottleneck for the system. It is quite obvious that if, for example, the index stage produces no more than 30.000 indexed tuples per second, then no matter how many joiners we are going to put into the system we are not going to get more that 30,000 joined tuples per second.

So for the first test we are going to create 10 indexer nodes. Using 10 nodes we are going to be able to cache all the B+ tree nodes and distribute the indexing process evenly between the nodes.

We have measured that the maximum throughput of the indexers in that configuration is up to 700,000 joinable tuples per second. That is in a range of stream keys up to 300,000,000.So that would be the maximum througput of the joiner stage. To measure the scalability of the system we are going to use 2, 4, 8, 16, 32, joiner nodes.

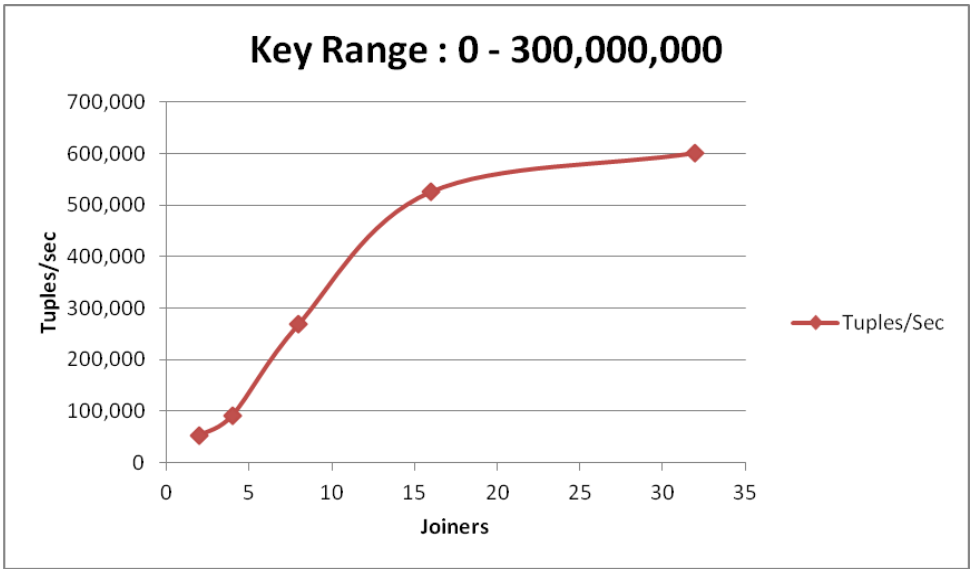

Figure 8: Joiner Scalability

As you can see we can achieve linear scalability up to the number of 16 joiners. That's because with this experiment at the point where the joiners where 16 the hit ration of the joiners node hit almost 99%. So it is logical that we not going to get much more performance even if we add more nodes. Eventually we would reach a maximum number higher that 600,000 tuples per second but the scalability will be far from linear from that point on. The most performance we are going to extract out of it will be from increased cpu resources.

The next measurements is going to have a stream tuple key range from 0 to 766,499,995 in order to see if that scalability pattern still holds.

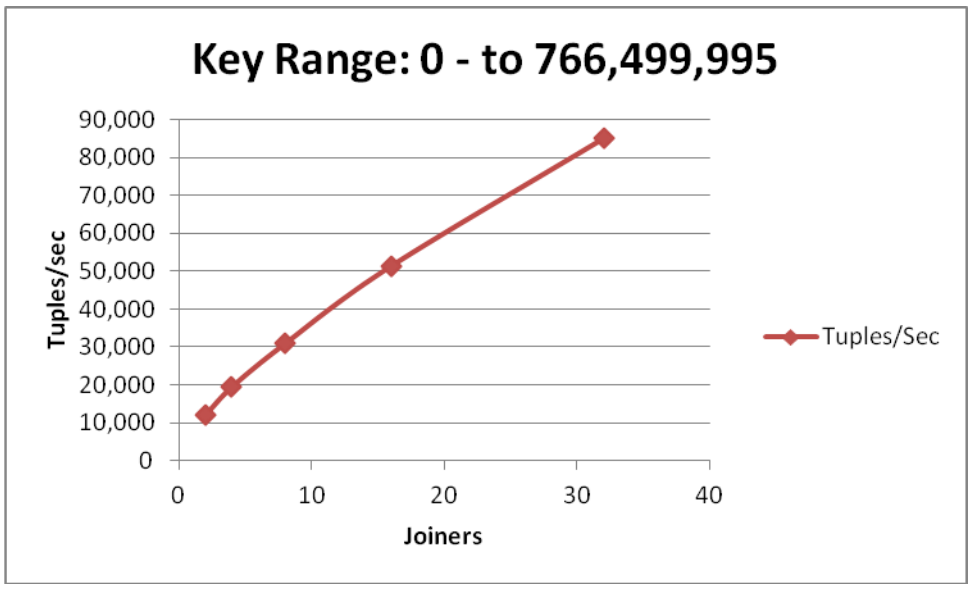

Figure 9: Joiner Scalability

As you can see the perfomance has seriously degraded, but the scalability still holds a linear scheme. That is reasonable as up unitl the 32 joiner we had almost 42% hit ratio. So adding more nodes to this well be very resonable and we would continue to take more linear scalability up until a certain point where we cannot extract much more performance out of the memory resources.

For the next experiments we are going to measure joiner scalability. We are going to rn tests with a stream with key range from 0 to 766,499,995 and we are going to test on 2, ,4, 8, 16, 32 joiner nodes. For that experiment we are going to measure how many recods per second the joiners produce without having to send them to the join router.

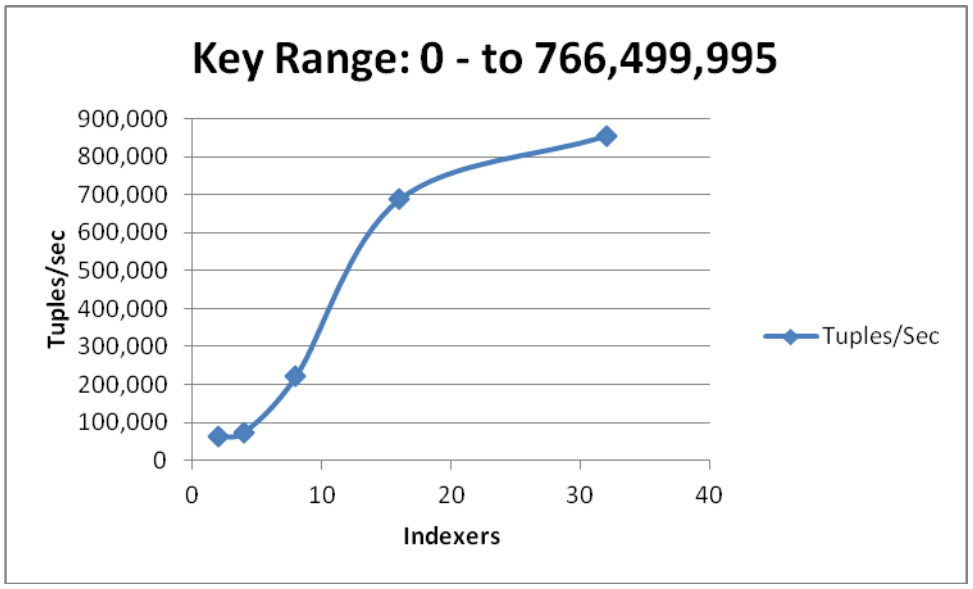

Figure 10: Indexer Scalability

As you can see we get linear scalability in the area from 10 to 16 joiners. From that point on the cache hit ration for all the joiner reached 99% percent and so the scalability figure drops. There is also an interesting result here. Between 4 and 8 indexers there is not much performance improvement. As you can see from Figure 10 they have almost

identical results. That has to do probably with the distribution of search in the tree. For some reason it was not distributed as equally as possible for these numbers. Also one more reason could be that we reached a very high percentage of hit ratio even from 4 indexers (it was more that 80% already)

On the next graph we are going to show you the same experiment with a stream with key range from 0 to 300,000,000.

As you can see the performance of the system quickly reaches a peak, and that's only natural as even from the 4 joiner we have reached almost 87% cache hits. We have a huge performance leap in 8 joiners where we almost reach a 97% hit ratio. From that point on the performance improvement is not as impressive. The reason is simply because we cannot extract more performance out of our memory resources because we have cached most of the tree nodes in our memory.

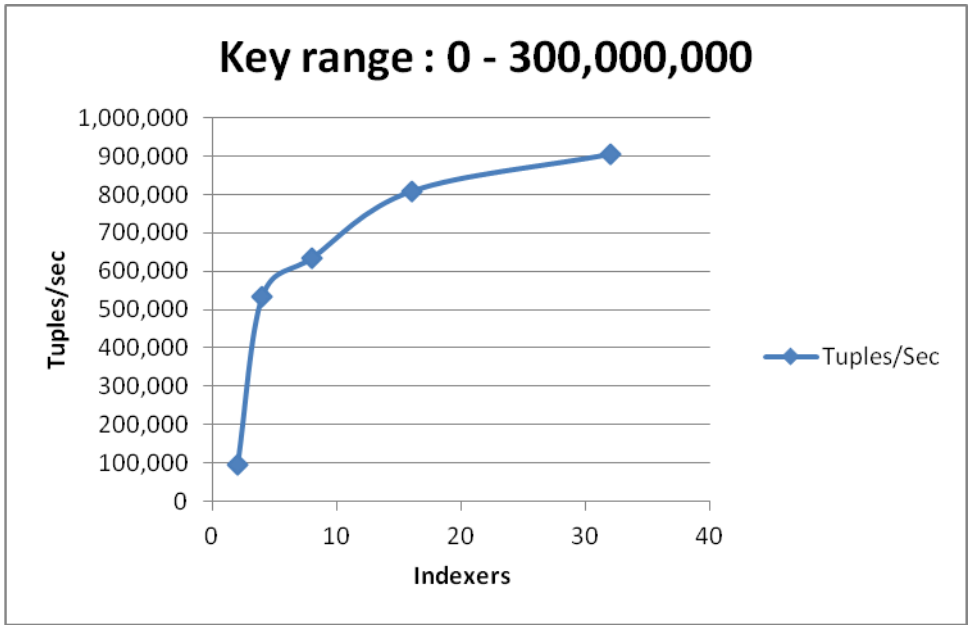

Figure 11:Indexer Scalability

5.3 Elasticity and Flexibility

In this section we are going to talk about how the elasticity of our design performed during our experiments and during our development. This is probably a good point to remember that the initial plan of our architecture was to add and remove resources on the go depending on the nature and the characteristics of the stream traffic. As we discussed, we could have several criteria on whether to add or remove computing resources.

We could potentially also include a cost formula for that matter. For example if we have a strict budget for that specific system in our pipeline we could potential calculate on the fly the cost we are putting down every moment and either decide whether we are able to add more resources or not. As a proof of concept we decided to use cache hit ratio as the performance criterion on whether to add or remove resources.

In that case we gave more attention to the Joiner nodes. As you have seen in our experiments because we have decided to give more memory resources to each joiner node there was no need most of the time to add more nodes. So most of the attention on the elasticity was on the joiner nodes.

The main issue with adding or resources is that it took a significant amount of time to deploy and run the containers. So we decided to use a separate thread to spawn the new nodes while the other nodes continue their job. When the new node is up and running he can enter the action as well with the algorithms we described in the previous sections.

We conducted the experiments for this as follows: we started with 5 joiner nodes. The join route was given a certain amount of cache hit percentage as e minimum requirement. We have created and algorithm that adjusts the velocity that we add resources depending on how far we are from the target. So for example in the case where we have a stream with tuple key range from 0 to 766,499,995 and we give a hit percentage target of 50% when we first start the system, the 5 joiners where capable of a hit percentage of 8%. That is very far away from the desired target. In that case we are more aggressive on adding more nodes. So at this rate we add almost one node every 40,000 records. We need to wait for that period of time in order to see whether we are getting close to our target and in order to do that we need to let the system normalize its behavior, and also it is possible that the stream traffic changes a bit so we wait to see whether we actually need to add more. But since we are so far behind from the target we don't want to wait too much either. In the above experiment starting from 5 indexers and 8% of cache hits, at the point where the system has added 38 nodes we were still at 23%. Now because we're getting closer to the target we can slow down a bit and add a node every 80,000 thousand tuples. After adding 50 nodes we were at 32%. So we slow down some more. After adding 53 nodes we hit 46%. At this point we stopped adding resources. We are too close the target. When the traffic normalized the 52 nodes where capable of reaching 70% cache hit ratio and it kept increasing. That is very natural as we were very aggressive on adding resources because we wanted to get up to 50% as fast as possible.

At the point where the system reached a 40-60% hit ratio area we were in a safe space. It is not always possible to achieve exactly the required target because every node has a specified amount of RAM and this total amount for ram is not always divided integrally with the amount of the blocked that need to be cached in order to achieve that percentage. So we have that area of +-10% to let the system cool down, normalize and see the nature of stream that is passing at the moment. We don't want to add or remove resources in that area as we are going to be trapped in a vicious circle of adding and removing constantly resources..

But in our case after approximately 10 minutes we have added 52 joiner nodes. And we have reached more that 70% of hit ratio. That meant that we are way over the minimum requirements and that we should probably start removing some nodes and see again where the system stabilizes. Because we are very close to our target ration the velocity with which we remove nodes at this point is very small. We give the system some time to cool down and reevaluate the passing traffic and the decide whether it is beneficial to remove more nodes.

After removing one node (now we are at 51), the hit ration was still climbing but not buy much. After removing 7 more nodes (45) it stopped rising and was now dropping but it was still around the 70% area. After removing 3 (42) more nodes it started to fall from 70%, but still at this point when the traffic normalized we were still at the 70% area. Because when we suddenly remove a node the hit ratio falls immediately. So it needs some time to cool down to see the actual hit ratio when using that many nodes. And the truth is that we are not at all aggressive at removing nodes, because we don't want to drop from the achieved target. But we don't want to be too lazy either because the whole point is to save costs.

So after approximately 15 minutes we were using 40 joiners the hit percentage was starting to stabilize at around 69 % percent. Now the velocity of removing nodes was very slow. As it is already evident we don't really want to remove nodes that fast and we favor the case where we are over the target ration rather than under. As you can see most of the time of the operation the system is over performing. If we are in a very strict budget we could alternate that logic and let the system be in favor of the cases of under-performance.

After about 30 minutes of the experiment running the 40 left joiner nodes where achieving around 69-68% percent of cache hits and the whole system was stabilized around those number. So to sum up this experiment, at the beginning we aggressively allocate more nodes in order to reach and overcome our target hit ratio as fast as possible. Then as the system cools down, the router realizes that we are over performing by a big margin. So he lazily starts to remove some joiners. After removing some he becomes more and more lazy on removing nodes. That is because he always favors over performance. After a while and after removing 12 joiners in total the system stabilizes in an over performing hit percentage.

During this whole time the auto scaling features of Kubernetes and Google Cloud where working well. Because Google Cloud also favors over performance in the peak of this experiment 60 VMs where allocated by the cluster management system where almost each one of them was hosting a single container. As the number of container was dropping so did the number of VMs and so did the total CPU usage.

At this point it is worth adding a couple of notes about the systems flexibility. We have analyzed in great detail the granularity of the working nodes. Router nodes are not aware on whether the working nodes are threads, or JVMs/processes or containers or whatever. All those details are abstracted away in our implementation. That means that the routers could potentially choose a worker node from any one out of these three implemented pools. This adds great flexibility to our system. For instance if you cannot afford to add another VM in your cluster but you want an extra boost of performance you can add some thread workers. If you can see that you don't have the scaling you want you then allocate a process worker that will run on its own JVM and will have its own heap and his GC pauses will not interfere with the system at all.

Another advantage of the flexible design is that you can be even more accurate with your performance targets. When running your nodes locally with either threads or processes you can dynamically adjust the memory buffers that those workers use instantly. For example if you need to add another 4550 buffer memory blocks to achieve a hit ratio of 52% then using local thread worker or process workers you can achieve that by adding worker with exactly that amount of buffers. Adjusting the memory buffers on the VMs is not that easy and it would require a certain protocol to achieve that which means more communication overhead.

We consider this flexibility a major advantage of our system. One can use this framework to implement very complex cost efficient policies according to his exact needs on several performance aspects, but always taking into account the budget deposits for the operation of this specific system.

6. CONCLUSION

In this research we combined two programming paradigms. We took the original idea of [1], that developed a single node solution for Joining Stream data with disc-stored relations near-real time. In order to keep up with the increasing traffic and data volumes we decided to transplanted it in a fully fledged elastic distributed system. The goal of this system is to provide real time performance for joining stream data with relational databases. Additionally this systems is capable of adding and removing computing nodes according to the nature and the volume of the incoming stream traffic as well as the size of the stored relation.

We developed algorithms that :

1) First of all, distribute equal portions of stream traffic to every worker node over a certain amount of time.

2) Need to be fair in terms of the amount of I/O that makes every worker node to perform.

3) Use all the available memory resources of the worker nodes as efficiently as possible. Try not to allow duplicate bocks to be cached by different worker nodes.

As far as the elasticity and flexibility of the system, at the beginning we aggressively allocate more nodes in order to reach and overcome our target performance aspects as fast as possible. Then as the system cools down, the router nodes might realize that we are over performing by a big margin. So we lazily start to remove some worker nodes. After removing some he becomes more and more lazy on removing nodes. That is because we always favor over-performance.

We've seen that with our algorithms and task distribution we achieved liner scalability for as long as we can increase the cache hit ration of the worker nodes. Finally the system performed well, overcoming by a big margin the performance of the single-node SSIJ solution.

REFERENCES

[1] Mihaela A. Bornea, Antonios Deligiannakis, Yannis Kotidis,Vasilis Vassalos, Semi-Streamed Index Join for Near-Real Time Execution of ETL Transformations, *Proceeding ICDE '11 Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, 2011, pp.159-170.

[2] A. Chakraborty and A. Singh, “A partition-based approach to support streaming updates over persistent data in an active data warehouse,” *IEEE Intern. Symposium on Parallel and Distributed Processing*, 2009.

[3] D. Burleson, “New Developments in Oracle Data Warehousing,” 2004.

[4] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica, Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad, Distributed data-parallel programs from sequential building blocks, *EuroSys ’07*, 2007.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82*, EECS Department*, UC Berkeley, 2011.

[7] James Turnball, The Docker Book, 2014.

[8] Docker guide*, https://www.docker.com/,* 2016.

[9] Kubernetes guide, http://kubernetes.io, 2016

[10] Google Cloud Platform, https://cloud.google.com/, 2016