NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS
POSTGRADUATE PROGRAM
COMPUTER SYSTEMS TECHNOLOGY

MASTER THESIS

DAB Join: A Distributed Adaptive and Balanced N-way Stream Window Join for Shared Nothing Clusters

Christoforos Svingos

Supervisor: **Yannis Ioannidis,** Professor

ATHENS

ΟΚΤΩΒΡΙΟΣ 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
”ΤΕΧΝΟΛΟΓΙΑ ΣΥΣΤΗΜΑΤΩΝ ΥΠΟΛΟΓΙΣΤΩΝ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσαρμοστική Κατανεμημένη Παραθυρική Ζέυξη Πολλαπλών Ρευμάτων σε Υπολογιστικό Νέφος

Χριστόφορος Σβίγγος

Επιβλέπων: Ιωάννης Ιωαννίδης, Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2016

MASTER THESIS

DAB Join: A Distributed Adaptive and Balanced N-way Stream Window Join for Shared Nothing Clusters

Christoforos Svingos
R.N.: M1358

SUPERVISOR:
Yannis Ioannidis, Professor

EXAMINATION COMMITTEE:
Yannis Ioannidis, Professor
Yannis Kotidis, Associate Professor

ATHENS

ΟΚΤΩΒΡΙΟΣ 2016

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Προσαρμοστική Κατανεμημένη Παραθυρική Ζέυξη Πολλαπλών Ρευμάτων σε Υπολογιστικό Νέφος

Χριστόφορος Σβίγγος
Α.Μ. M1358

ΕΠΙΒΛΕΠΩΝ:
Ιωάννης Ιωαννίδης, Καθηγητής

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:
Ιωάννης Ιωαννίδης, Καθηγητής
Ιωάννης Κωτίδης, Αναπληρωτής Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2016

ABSTRACT

In this work, we present DAB Join, an adaptive operator that enables scalable processing of Multiway Windowed Stream Joins using a shared nothing cluster. DAB join (1) supports any kind of join predicates, (2) minimizes the network cost while at the same time distributes the load equally to all cluster nodes, and (3) uses only one hop to distribute the data and execute the join, avoiding the distribution of intermediate results that may be very large. We have implemented DAB Join on top of an experimental system named Exastream, which supports the distributed execution of jobs expressed as DAGs of user defined operators. Based on synthetic streams, Our experimental results show that our algorithm is scalable. Additionally, DAB Join adapts to changes of stream input rates, which results in better execution times compared to non-adaptive alternatives.

SUBJECT AREA: Computer Science

KEYWORDS: stream processing, distributed join, window join, adaptation,

ΠΕΡΙΛΗΨΗ

Σε αυτή την δουλειά, θα παρουσιάσουμε έναν προσαρμοστικό αλγόριθμο για παραθυρική ζεύξη σε πολλαπλά ρεύματα δεδομένων χρησιμοποιώντας υπολογιστικό νέφος. Τα κύρια χαρακτηριστικά του αλγορίθμου είναι ότι (1) υποστηρίζει όλων τον ειδών τα κατηγορήματα, (2) μειώνει το κόστος μεταφοράς ενώ την ίδια στιγμή διανέμει ισότιμα τον φόρτο σε όλους τους κόμβους του υπολογιστικού νέφους, (3) χρησιμοποιεί μόνο ένα βήμα για να διανείμει τα δεδομένα και να εκτελέσει τη ζεύξη, αποφεύγοντας την διανομή των ενδιάμεσων δεδομένων τα οποία μπορεί να είναι τεράστια. Υλοποιήσαμε τον αλγόριθμο ζεύξης σε ένα πειραματικό σύστημα με το όνομα ExaStream. Το ExaStream υποστηρίζει την κατανεμημένη εκτέλεση δουλειών οι οποίες μπορούν να εκφραστούν σε ακυκλικούς γράφους από διεργασίες που έχουν οριστεί από τον χρήστη. Τα πειράματα έγιναν σε συνθετικά δεδομένα και δείχνουν ότι ο αλγόριθμός μας κλιμακώνει ενώ ταυτόχρονα προσαρμόζεται σε αλλαγές της ροής των εισερχόμενων ρευμάτων. Λόγο της προσαρμοστικότητας ο αλγόριθμός μας συμπεριφέρεται καλύτερα και δίνει καλύτερους χρόνους εκτέλεσις σε σύγκριση με μη προσαρμοστικές εναλλακτικές.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Πληροφορική

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: επεξεργασία ρευμάτων, κατανεμημένη ζεύξη, ζευξη παραθύρων, προσαρμοστικότητα,

Αφιερώνω την εν λόγω εργασία στην οικογένεια μου.

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω το οικογενειακό και φιλικό μου περιβάλλον για την υποστήριξη τους κατά τη διάρκεια του Μεταπτυχιακού πρόγραμματος και κατά τη διάρκεια εκπόνησης της εν λόγω εργασίας.

Ιδιαίτερες ευχαριστίες θα ήθελα να δώσω στον Θεόφιλο Μαΐλη για τις διορθώσεις στο κείμενο.

Επιπλέον, θα ήθελα να ευχαριστήσω θερμά την ομάδα του Exareme για την άριστη συνεργασία και τις πολύτιμες γνώσεις που μου προσέφεραν.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

A fundamental problem in Distibuted Stream Management Systems (DSMS) is processing of continuous queries over high-volume and time-varying data streams under resource constraints. Our research, focuses on multi-way windowed stream join, a core operator in DSMS systems. The window stream join operator can be used to discover correlations across different streaming sources, which has many important applications in video surveillance, network intrusion detection, and sensor monitoring.

A typical scenario, requires diagnosing and monitoring of powergenerating turbines. In the described scenario, several service centres are dedicated to diagnosing of over than 100,000 thermocouple sensors installed in 950 power generating turbines located across the globe. One typical task of such a centre is to detect in real-time potential faults of a turbine caused by, e.g., an undesirable pattern in temperature's behaviour within various components of the turbine. This task requires to extract and correlate streaming data produced by up to 2,000 sensors installed in different parts of the turbine. Such kind of tasks needs to execute expensive multi-way window joins over streams.

This need has triggered the design of scalable approaches that provide low latency answering to queries on high-velocity live streams. In this work, we introduce DAB Join, an adaptive operator that enables scalable processing of multi-way windowed stream joins, using a shared nothing cluster. DAB join extends the *Symmetric Fragment and Replication Algorithm* [9] to support window-stream joins that adapt to different stream rates. The main characteristics of a DAB Join is that it (1) supports any kind of join predicates (2) minimizes the network cost while at the same time distributing the load equally to all nodes of the cluster (3) uses only one hop to distribute the data and execute the join, avoiding to distribute intermediate results that may be huge.

To put our ideas into practice, we have developed ExaStream, an distributed realtime computation system. Exastream supports the distributed execution of jobs that are expressed with DAGs of user defined operators. Our experimental results, based on synthetic streams, show that our algorithm is scalable. Additionally DAB Join adapts to changes of stream input rates. This results to better execution times compared to corresponding non-adaptive alternatives.

The rest of the thesis is organized as follows. Section 2 presents the related work, Section 3 shows the problem formulation, Section 4 presents the Theoretical Model of our algorithm, Section 5 presents our algorithm, Section 6 presents the implementation, Section 7 presents our experimental evaluation and final section 8 presents our conclusions.

2. RELATED WORK

In [2] introduced Eddies. Eddies are among the first adaptive techniques known for query processing. Eddies act as a tuple router that is placed at the center of a dataflow, intercepting all incoming and outgoing tuples between operators in the flow. Eddies observe the rates of all the operators and accordingly makes decisions about the order at which new tuples visit the operators. In principle, Eddies are able to choose different operator orderings for each tuple within the query processing engine to adapt to the current information about the environment and data.

In [4] proposed two state replication based distributions for generic N-way window join operators: aligned tuple routing (ATR) and co-ordinated tuple routing (CTR). Both algorithms have make the assumption, that all streams will pass through a Diffusion Operator that runs on one Server.

ATR picks one input stream as the master stream and partitions the master stream among the processing nodes. All the other slave input streams are distributed to the processing nodes with some overlaps, to ensure the semantics of the window constraints. In ATR the segment length is an important parameter for the load diffusion. However, the ATR approach works under the condition that the window constraint (W) are much greater than the size of each part (T). When W is comparable with T, the memory waste and redundant computation can be significant.

CTR is a multi-hop semantics preserving tuple routing where intermediate join results are transferred among nodes during each hop. A weighted minimum set covering is utilized to identify the optimal routing for each tuple to “find” all correlated states. In CTR the number of redundant state is determined by the minimum set covering at runtime. CTR faces the following dilemma. The more redundant states, the smaller set covering may exist. Then the incoming tuples will be stored in fewer nodes, which may make future set covering large. More seriously, the states in CTR may converge to one (or a small subset of) node if sometime only one copy of the input tuples is stored in the cluster, since the future set covering will direct all later tuples to that node. Then no distribution is achieved.

The authors of [11] introduced the Pipelined State Partitioning (PSP) that focuses on distributed processing of generic N-way window joins with arbitrary join predicates, especially for use cases with expensive join probing cost over large window constraints. This work is related to the state-slicing concept presented in [12]. PSP splits the window time into N disjoint slices, where N is the number of Nodes in a shared nothing cluster. Next PSP organizes the nodes in a pipelined state-slice join ring. The probing tuple and intermediate results traveling through the ring. This means that tuples pass through all cluster nodes at least once. In more detail, the worst case (M-1)N hops of intermediate propagations are needed for an M-way join.

In [8] the Flux operator is presented. Flux is a general adaptive operator that encloses

adaptive state partitioning and routing. Although the authors focus on single-input aggregate operators, it can support a restricted class of join predicates, e.g. equi-join. Flux supports equi-joins under skewed data settings but requires explicit user knowledge about partitions before execution.

Finally, in [3] the authors use the Symmetric Fragment and Replication Algorithm [9] to support adaptive online joins in relational static tables. The problem they tried to solve was finding the optimal replication factor for all relations without having statistics beforehand.

3. MODEL AND PROBLEM DEFINITION

In this work we tackle the problem of scalable processing of multi-way window joins. We assume that the timestamps of stream tuples are globally ordered. Windows define the scope of the otherwise infinite streams for stateful operators. In this section we will provide some preliminaries on streams.

A data stream S consists of a sequence of tuples $s \in S$, such that each tuple caries a timestamp $s.t$ in its header to denote the time when the tuple s arrives at the stream S. For stream element arrivals we assume a time domain T. We can slice a stream into infinite disjoint sets with size T, and map each disjoint set with an unique ordered id. So $X_i[t_s, t_e]$ defines a set with $id = i, i \in \mathbb{N}$, which is responsible for time units t for which $t_s \leq t \leq t_e$ and $t_e - t_s = T$. We define as window $WD_i[t_e - W, te]$ with duration W and $id = i$ all tuples that arrives in time t for which $t_e - W \leq t \leq t_e$ and t_e is the time that defines the end of set $X_i[t_s, t_e]$.

So we can say that T defines the frequency and W defines the time duration of a window. For example we can say that every 10 seconds ($T = 10$) we will look back 15 seconds ($W = 15$). This example defines sliding windows that overlaps for 5 seconds. To define non overlapping windows we can say that for each 10 seconds we will look back 10 seconds. Notice that if two streams have the same T then we can give the same id to windows that reference to the same time durations, simply map this windows with the epoch time of t_e.

From the above definition of the window, we can define the window join as a simple equi-join between window ids. For example, suppose that we have streams S_1, S_2 that both have the same T and W. Then we can take the window join of both, simply executing the constrain $S_1.wid = S_2.wid$. Notice, that our model can join a current window of stream S_1 with windows that is behind in time from stream S_2. For example, the constrain $S_1.wid = S_2.wid - 3$ joins the current window of S_1 with the window that came 3 windows behind.

4. SYMETRIC FRAGMENT AND REPLICATION ALGORITHM

A lot of work has been made during the last years towards answering distributed equi-join queries between two relations [6] [7]. But what happens when we want to execute theta joins, non-equi joins or equal joins with data skews or N-way joins? The Fragment and Replicate Algorithm is an algorithm that supports such kind of joins. In this section, we will make a report of this algorithm, as it is referenced in [9], and we will introduce a different perspective for the same problem.

4.1 Stamos And Young Perspective

In this section, we will describe the Symmetric Fragment and Replicate Algorithm as described by Stamos and Young. Suppose that we have six join sites and we want to join relations R and S. Relation R is partintioned into two disjoint fragments (RF_1 and RF_2) and relation S into three disjoint fragments (SF_1, SF_2 and SF_3). The rectangle in figure 1 shows the six join sites. We can take the join of R with S if we replicate each fragment of R across one row of join sites and each fragment of S across one column of join sites. Each site is responsible to execute a local join algorithm for his fragments of data.

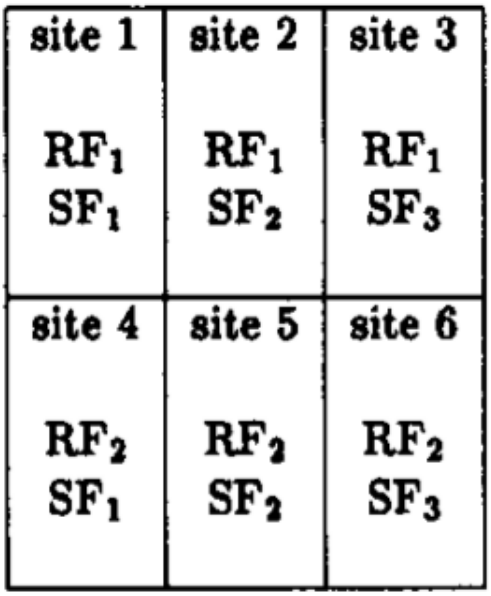

Figure 1: Join Cites for SFR Algorithm and relations S and R

Suppose now that we ignore any initial replication of tuples. Then for the same problem there are 4 ways to arrange the sites into a rectangle: box(2, 3), box(3, 2), box(6, 1) and box(1, 6). So now the problem is an optimization problem. We must define the best rectangle that minimize the replication cost.

Now that we get a notion about the problem, we can formalize the SFR algorithm for a N-way join. We use the following notation to simplify our discussion:

S : the number of join sites
N : the number of relations involved in the join
Ri : The relation i, i belongs to 1, ..N
Li : The length of the box in the ith dimension
Mi : The size of the ith relation

Let $box(L_1, L_2, \ldots, L_N)$ be an N-dimensional box with volume S. Consider the site located in $cell(p_1, p_2, \ldots, p_N)$. This site multicasts its fragment of R_1 to all sites whose location is of the form $cell(p_1, *^{(N-1)})$. The site multicasts its fragment of relation R2 to all sites whose location is of the form $cell(*, p_2, *^{(N-2)})$. In general, the site multicasts its fragment of relation R, to all sites whose location is of the form $cell(*^{(i-1)}, p_i, *^{(N-i)})$. The degree of replication of relation R_i is S/L_i.

So, if we have S sites to perform an N-way join, the problem is to determine the shape of the N-dimensional box with volume S that minimizes communication. When replicating relation R_i, we make S/L_i new copies of R_i and send each copy to a different site. The communication cost of replicating R_i is $M_i * S/L_i$. Therefore, the total communication cost of the SFR algorithm for a N-way join is:

$$optimalcost = \sum_{i=1}^{N} M_i S/L_i$$

Our goal is to minimize the cost subject to the constraints that L_i's are positive integers and

$$\Pi_{i=1}^{N} L_i - S = 0$$

However the authors cannot find a good solution to solve the above equation. In the paper suggesting a heuristic solution that does not find always the optimal solution for the problem. More specific in the appendix the authors says:

> "Our colleague Nimrod Megiddo developed a dynamic programming algorithm that finds the optimal integer solution. Because its running time is exponential, we searched for more efficient ways to find a good integer solution. In conjunction with Nimrod Megiddo, we developed a simple algorithm that finds the optimal integer solution when the number of join sites is the power of a single prime."

In the next sections we will define the problem in an our perspective and we will give an algorithm that finds the best integer solution.

4.2 Our Perspective

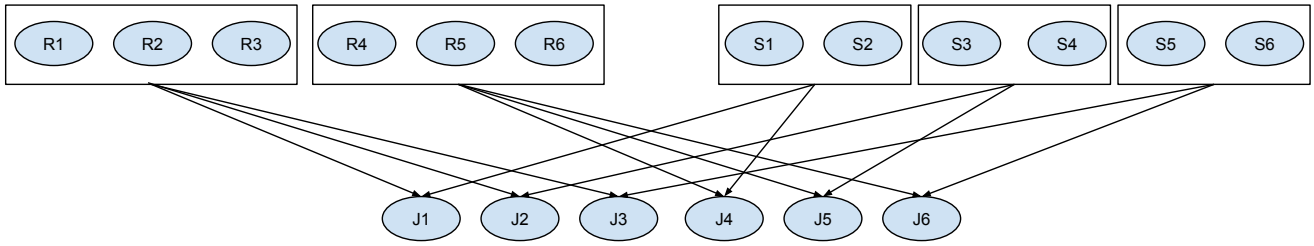

Figure 2: SFR Algorithm for relations S and R using map-reduce jobs

Suppose, that we have again 6 join sites and the relations R and S are partitioned in 6 parts each. One solution to take the R, S join is to create two groups for relation R (three partition each group) and three groups from relation S (two partitions each group) and then get all combinations for these groups as shown in figure 2. We define this topology as mapping(2, 3).

Figure 3 shows all combinations for mapping(2, 3) in a tree representation. The leafs of the tree represent the join sites. For the same problem, we can define the mapping(1, 6), mapping(3, 2), mapping(2, 3), mapping(6,1). All mappings generated from the results of the equation x*y = 6, where x, y > 0 and x is the number of R groups and y the number of S groups. Relation R is replicated three times and relation S two times, as it is shown on Figure 3. This occurs, if we find in how many leafs participate one group of each. For example for relation R we get 6/2 = 3 replicas and for relation S we get 6/3 = 2 replicas.

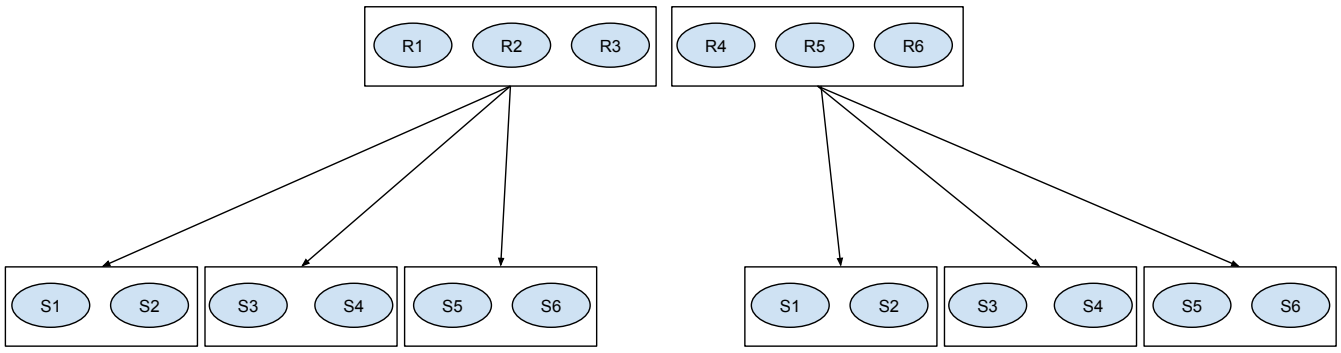

Figure 3: The tree representation of all compinations of groups from relations S and R

In the rest of this section, we define the replication pattern formally extend it to handle N-way joins. We use the following notation to simplify our discussion:

S : the number of join sites
N : the number of relations involved in the join
Ri : The relation i
Li : Number of groups for relation i
Mi : The size of the ith relation

The notation here is similar to that defined by Stamos and Young. The difference is that we define L_i as the number of groups of relation i. So, again, the problem is to find the $mapping(L_1, L_2, \ldots, L_N)$ that minimize the replication cost. The following equations defines these problem:

$$\Pi_{i=1}^{N} L_i = S$$

$$minimize(\sum_{i=1}^{N} M_i S/L_i)$$

These equations are the same as Stamos and Young. The two models are identical. Our representation is appropriate for map-reduce jobs and will help us in the next sections to define our adaptive join algorithm.

In listing 1 we describe the algorithm that enumerates all solutions for integer solution problem. The algorithm first adds to the solutions the mapping(1, 1, …, S). Then the algorithm executed in N-1 steps. Because we want the $\Pi_{i=1}^{N} L_i = S$ we initialize the L_N with all possible d, where $d \in$ {set with all divisors of S}, and L_{N-1} with the quotient of the S/d. Each distinct mapping added to a set with all solutions (Sol). Then we repeat the above steps for each $mapping \in Sol$, where S now is equals to $mapping[N-1]$. In N-1 step the algorithm finds all mapping solutions.

Listing 1: Integer Solution Algorithm

```
getDivisors(_s):
    allDivisors := {}

    for _d in {2, ..., sqrt(N)}:
        if _s % _d == 0:
            allDivisors.add(_d)
            allDivisors.add(_s / _d)

    return allDivisors


main():
    solutions := {}
    mapping = [1, ..., N]
    solutions.add(mapping)

    for _x in {N, N-1, ... 2}:
        for _map in solutions:
            for _d in {getDivisors(_map[_x])}:
                mapping[_x] = _d
                mapping[_x-1] = _map[_x] / _d
                solutions.add(mapping)
```

For better understanding suppose that we have the equation:

$$x * y * z = 8$$

The first iteration of the algorithm give us the solutions: mapping(1,1,8), mapping(1,2,4), mapping(1,4,2), mapping(1,8,1). In the next step the algorithm give the solutions: mapping(1,1,8) from mapping(1,1,8), the solutions: mapping(2, 1, 4) from mapping(1,2,4) the

solutions: mapping(2,2,2), mapping(4, 1, 2) from mapping(1,4,2) and the solutions: mapping(2,4,1), mapping(4,2,1), mapping(8,1,1) from mapping(1,8,1). And then the algorithm ends.

5. DAB JOIN

DAB join uses the Symmetric Fragment and Replication algorithm to distribute the windows to join sites. Because streams arrive with unpredictable input rates our algorithm dynamically adjusts to the different rates of streams. In this section we will describe our algorithm and we will give some implementation details of the DAB join algorithm.

5.1 Initial Phase

Suppose that we want to join streams A and B and we have four available CPUS. Figure 4 shows the topology that DAB join uses. The first level of the graph composed of operators named transponders and the next level with operators named joiners. Each stream has its own transponders operators. For example, the operators A_1, A_2, A_3, A_4 belong to Stream A. The main work of transponders is to broadcast their data to joiners and the main work of joiners is to locally execute the join. To simplify our model we assume that transponders have data distributed to them in a round robin manner.

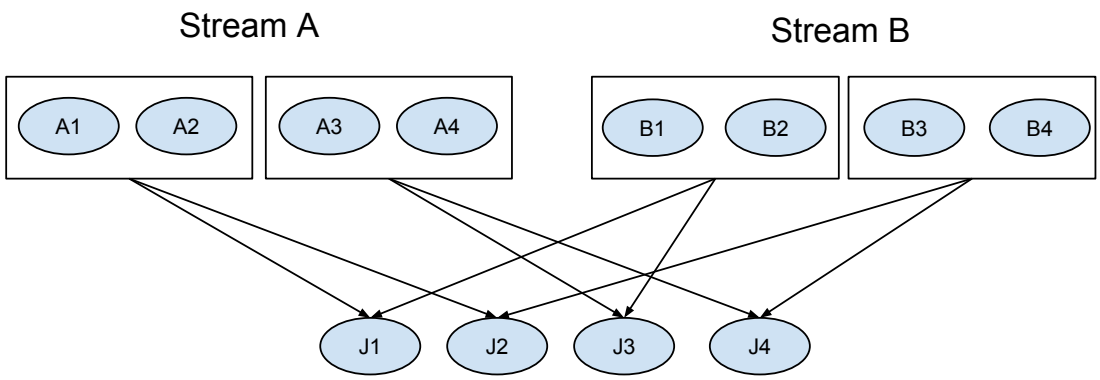

Figure 4: DAB Join Topology

In our example, stream A and B consist of 4 transponders each. Additionally, the same number of operators is occupied from joiners. The distribution of operators is not accidental. Transponders consume a minimum portion of the CPU because their work is only to distribute their contents. On the other hand, joiners consume much more CPU because they must execute the join. So in the corresponding topology we exploit better the available resources of the cluster.

As mentioned the main work of a transponder operator is to broadcast its content. The number of broadcasts that a transponder distributes its data to is determined by the Symmetric Fragment and Replication algorithm as mentioned in section 4.2. The number of broadcasts for each operator is the number of replicas of the stream that this operator exists. The mapping at the initial state of the DAB Join is the mapping that occurs if we assume that the rates of all streams are similar. For example in Figure 4 we use the mapping(2,2). That means that we have split each stream into 2 groups . For this mapping each transponder operator must make 2 broadcasts.

Because, we want the communication to be made asynchronously, each transponder keeps a buffer for each of its outputs. Each tuple produced from a transponder is written to N buffers, where N is the number of outputs for this transponder. So when a joiner reads from an output, then it reads from a specific buffer. If one of the joiners delays to read their output this doesn't delay the other joiners. If the buffer of one output is filled, then the transponder will block writing tuples to buffers until it finds the appropriate space.

Respectively, the joiners read asynchronously. This means that the tuples that the joiners read are stored in buffers. When the join algorithm needs more tuples then reads tuples from the transponders buffers. When the join algorithm is blocked (e.g for an io) then the joiner reads tuples from transponder and fills their buffer. This architecture design is necessary for performance reasons and as we see below affects our rebalance algorithm.

5.2 Rebalance Phase

As mentioned before, the rates that streams arrive may vary from one time point to another. If this change on input rates happens, then the best mapping for this input rates may be different. For example suppose that the rate of stream A and stream B from previous example was 10 tuples per second. The mapping for this rates is similar with the one that is seen in figure 4. Suppose now, that after a while the rate of stream A changes to 100 tuples per second. If the mapping topology remains unchanged then 2*100 + 2*10 = 220 tuples per second will be transferred. On the other hand, if stream A is replicate only one time, stream B 4 times (figure 5), then we transfer 1*100+4*20 = 180 tuples per second. So if our algorithm adapts to the stream rates in the specific setup we can transfer 20% less tuples to the network!

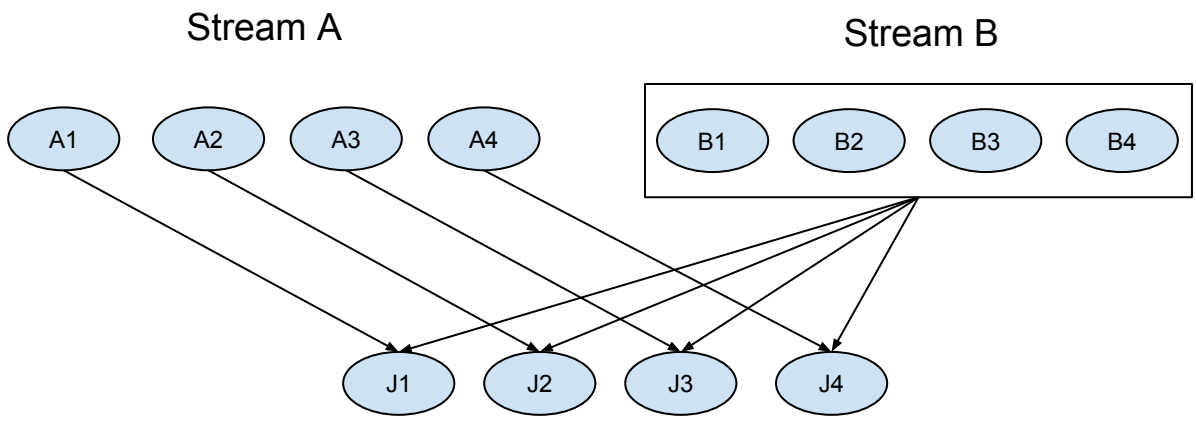

Figure 5: DAB Join Topology after Rebalance

As we said, the first constraint of our join is an equality constraint on window ids. This gives us the ability to change the mapping of DAB Join for the next window that none of its tuples has been processed yet. For example, suppose that all joiners have processed tuples for window with id 5 for streams A and B. Additional, joiners have not yet process

any of the tuples of window 6. In this case, we can rebalance the way that the joiners read their tuples from transponders without influencing the result of the join.

Our algorithm, exploits the fact that the tuples are partitioned in a round robin manner. This means that each joiner has the same view of the input rate of streams. Suppose that, we have windows with duration 1 second and rates of 40 t/sec for stream A and 4 t/sec for stream B. For mapping(2,2) each joiner will have for some window, 2 tuples for stream B and 20 tuples from stream A. If the joiners know the mapping they can divide with the replicas of each stream and calculates the exact rates. This means that each joiner can decide alone which window must change the mapping and will be sure that all other joiners will decide to change the mapping in the same window. So all joiners can decide at the close of each window if a rebalance may occur executing the integer solution algorithm.

However, if the number of total tuples in each window cannot be divided perfectly by the number of joiners, then some joiners may have one more tuple from others. On the other hand, if the number of tuples in each window is less than a threshold, some joiners may take no tuples for some window. To simplify our model, we assume that all joiners have the same number of tuples for each window. This can maintained, if the operators that make the round robin reshufle add dummy tuples to preserve this constraint.

Now that we know when a joiner decides to trigger a rebalance, we can describe the rebalance algorithm. When a joiner decide that a rebalance may occur, sends rebalance messages to all transponders of stream A and stream B that correspond to the new mapping. In our example, DAB join changes for mapping(2,2) to mapping(4,1). So joiner 2 will send rebalance messages to transponders A_2, B_1, B_2, B_3, B_4.

Table 1: Rebalance Message

State	The state is the number of rebalances that have occured
NumberOfBroadcasts	The new number of broadcasts that the transponder must support
PreviousOutput	The number of output that the joiner has read from the previous mapping
NextOutput	The number of output that the joiner will read when the rebalance occur

The rebalance messages contain the information presented in table 1. The State says how many rebalances have occurred from the beginning of the DAB join execution. This information is necessary because, in an extreme case we may receive a rebalance before the previous rebalance has been executed. The State ensures that the new rebalance doesn't start its execution before the previous rebalance has finished.

As we say each joiner reads a specific output of a transponder. For example, in figure 4 the joiner J_2 reads the second output of transponders A_1 and A_2 and the first output of transponders B_3, B_4. The NumberOfBroadcasts determines the new total number of outputs that a transponder may support. The PreviousOutput determines the number of

outputs that the joiner has already read and, the NextOutput determines the output that the joiner will read after the rebalance. Notice, that the NumberOfBroadcasts also determines the number of messages that this transponder waits to receive for this State. This is necessary for a transponder to decide when the rebalance has terminated.

Figure 6 shows the actions that take place when joiner J_2 trigger a rebalance. When a rebalance message arrives into a transponder then the transponder begin the rebalance phase. In this phase the transponder creates so many new buffers as the NumberOfBroadcasts says. More specific the new buffers are replicas of the old ones. The replicated buffer decided for the first message that the field PreviousOutput is not empty. Notice that joiner J_2, as shown in figure 6, will send messages to transponders B_1 and B_2, but joiner J_2 does not read any of the outputs from these operators. So the field PreviousOutput for these messages will be empty. The most important is that a message that the field PreviousOutput is not empty says that the joiner will not read tuples from this buffer again! We can't assume this for the other buffers, because a joiner may not have the chance to read yet the tuples that will trigger the rebalance.

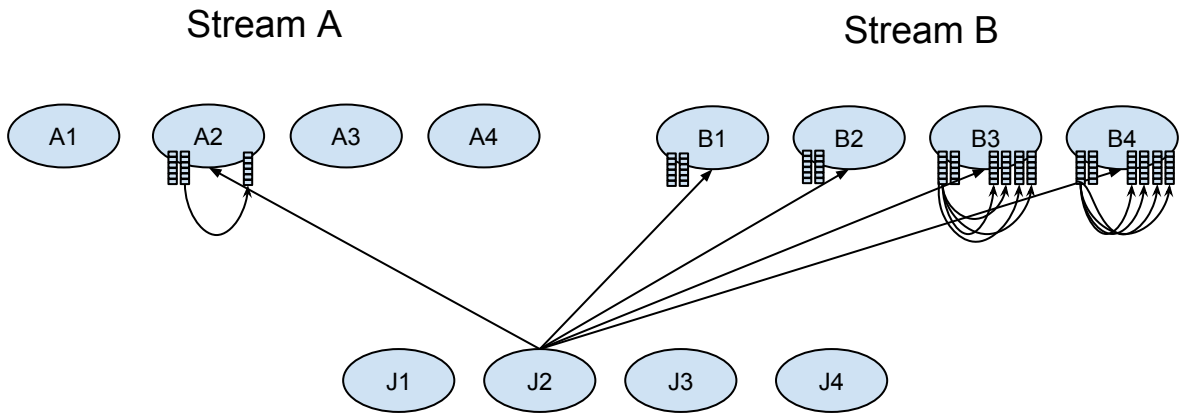

Figure 6: Actions that take place at rebalance phase

In our algorithm, joiners are blocked until they get a positive answer from all rebalance messages that have been send. For example, joiner J_2 will get answers from transponders A_2, B_3, B_4 when the initialize of new buffers finished. But the answers from B_1 and B_2 will occur when these transponders initialize their own buffers. This will happen, if these transponders get a rebalance message from another joiner that the field PreviousOutput is not empty.

When the rebalance terminates the joiner will start to read tuples from the new outputs. In the next section, we will describe in more detail the transfer phase.

5.3 Transfer Phase

As mentioned, at the rebalance phase the transponder will replicate one of its buffers as many times as defined in the NumberOfBroadcasts field. The transfer of data becomes

asynchronously and in batched manner. This means, that a joiner may read tuples from windows that belong to time after the window that will trigger a rebalance. These tuples are saved into the joiners buffer and removed from the transponder's buffer. When a rebalance occurs, the joiners empty their buffers, and continue to read data from transponders using the new mapping. This means, that our algorithm as being described, is incomplete.

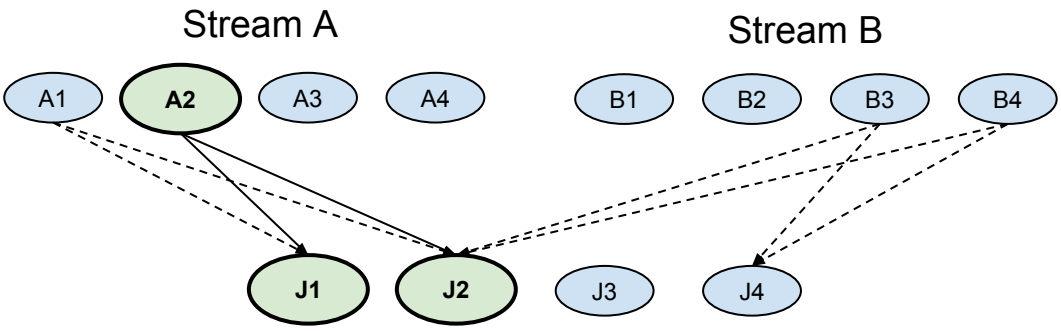

Figure 7: Transponder's A_2 outputs

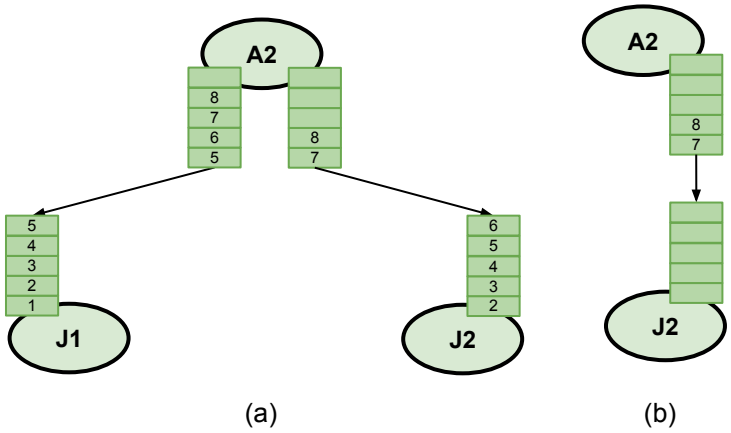

Figure 8: Buffers state before and after rebalance.

Figures 7, 8 show an example that describes the aforementioned problem. Figure 7 shows that joiners J_1, J_2 read the first and the second output of transponder A_2 respectively (mapping(2,2)). Figure 8a shows a possible state of the buffers for transponder A_2 and joiners J_1, J_2, where buffer numbers correspond to the number of windows that exist in each buffer. Suppose, that J_2 decides to trigger a rebalance, with a (4, 1) mapping, for window with id 1. After the rebalance completes, J_2 will read data only from transponder A_2. As displayed in figure 8b, joiner J_2 will empty its buffer and will request for data from transponder A_2. This leads to a waste of data and unsound results.

To solve these problem, joiners read data with a semi pull-push model. Each joiner, makes a request to get the data from the output of a transponder. The transponder responds with data whose length is equal to the length of the buffer. The joiner adds the data to its own buffer and continues to read with a new request. If the size of the joiner buffer equals the

size of the transponder buffer then in any case we have stored data to the buffer which have been derived from at most the buffers of two requests.

To better understand this, let's study an extreme case. Suppose that the buffer in the joiner side is full. The latter means that the joiner does not make another request until it reads all data from the served request and puts them to the buffer. The joiner then reads a tuple from the buffer and decides that it must trigger a rebalance. Before it triggers the rebalance the joiner may make another request to the transponder, since there is room to the buffer.

In this example, we have read tuples equals the size of one buffer and we make another request to get the rest of tuples with length equal to the transponder buffer length. If we trigger the rebalance then the data that have been transferred to the joiner and belong to a window with bigger id from the one that triggers the rebalance, will not be processed. For this reason, the transponder stores to a "historic" buffer with size equals the size of two buffers the data that were transferred to the joiner side. So when a rebalance occurs the transponder re-transfers the data that were stored into the historic buffers. Especially, for rebalance messages we send as parameter the window id that triggered the rebalance. So the transponders after the rebalance phase re-transfer only the tuples that are greater than their window id.

5.4 Rebalance Constraint

As we said, our algorithm works only if the transponders accept at least one rebalance message that its field PreviousOutput is not empty. That means that after the rebalance the transponders must transfer data to at least one joiner that transfers data before. For example, suppose that the two states before and after the rebalance are displayed in figure 9. As shown in the new state the J_2 joiner requests for data the transponder A_3 while joiner J_3 requests for data transponder A_2. The transponders A_2 and A_3 never accept rebalance messages with the field PreviousOutput filled. These transponders will never finish the rebalance phase because they can not close and clone his buffers to the new ones. So our transition for one rebalance state to another must ensure the above constraint.

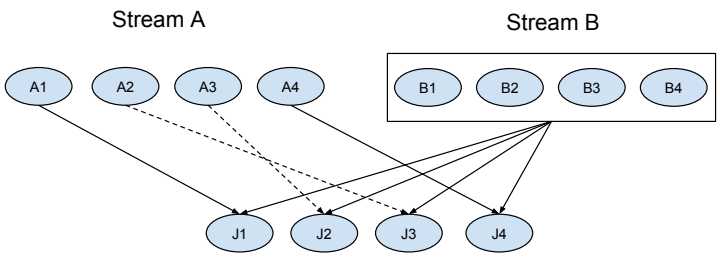

Figure 9: Problem at the rebalance phase

Suppose that we have the tree representation of figure 10. This tree represents the combinations of all Groups for streams A, B and C. In our example, we have 8 join sites and

the mapping(2,2,2). Each array in the figure, represents the transponders of a group. The first level of the tree has the groups of stream A, the second the groups of Stream B and the third the groups of stream C. Each group has the transponders in sorted order. For example if we have all transponders of Stream A in an array with 8 cells in sorted order and split this array then we take the Group 1 that has the transponders 1,2,3,4 and Group 2 that has the transponders 5,6,7,8. Respectively, the same algorithm follows to take the groups from the other streams. To take the combinations of all groups we can simply run a 3-way nested loop.

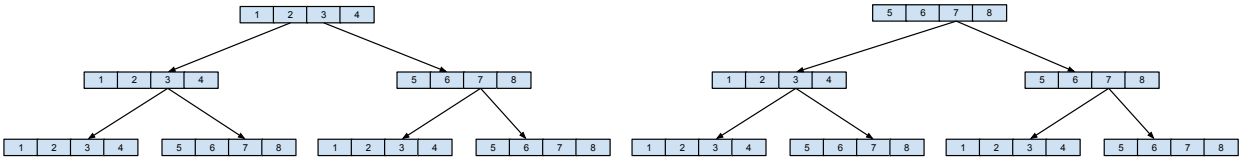

Figure 10: Problem at the rebalance phase

Suppose now that we want to change from mapping(2,2,2) to mapping(1,8,1). Follows the algorithm that we just describe we get the tree of figure 11. This tree shows that transponder 3 of stream B will get only one rebalance message from the 3rd joiner. But as we show from figure 10 joiner 3 didn't takes data from transponder B3 before! So we must rearrange the transponders for Stream B before we do the rebalance. One possible redeployment for stream B is the following:

$$1,2,5,6,3,4,7,8$$

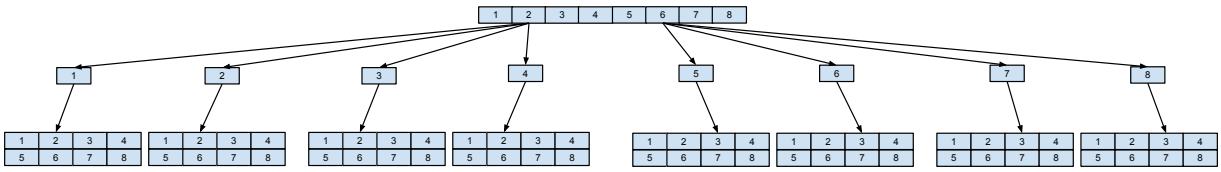

Figure 11: Problem at the rebalance phase

Suppose now that we have the notation of table 2. To decide when the transponders of one group need redeployment we create the following equation:

$$Size_{old}(Group(X))/Size_{new}(Group(X)) \neq JoinerSites_{old}(Group(X))/JoinerSites_{new}(Group(X))$$

To understand better what this equation says we will explain it using an example. Suppose

that we rebalance from mapping(2,2,2) to mapping(1,8,1). For this mappings we have:

$$\begin{aligned} Size_{old}(Group(A)) &= 4 \\ Size_{new}(Group(A)) &= 8 \\ JoinerSites_{old}(Group(A)) &= 4 \\ JoinerSites_{new}(Group(A)) &= 8 \end{aligned} \tag{1}$$

The equation result is false. This is reasonable because in first rebalance State the first 4 transponders of A take part to first 4 joiner sites, and in the next rebalance state all transponders of A take parts to all sites. So we reduce the number of groups with a factor of 2 but we duplicate the number of join cites that a group takes place. In the other hand the equation for stream B is true:

$$(4/1) \neq (2/1)$$

Indeed, the number of transponders in each group reduced at a factor of 4 but the join sites that the group takes part reduced for a factor of 2. To understand better this, let's see the path of B3 transponder before and after the mapping change. transponder B3 takes part to group 1 of stream B and participates to joiners 1, 2 and 5, 6. After the rebalance, the transponder B3 takes part to group 3 of stream B and participates to joiner 3. This happens, because, before rebalance we have 4 transponders of Group 1 to participate to 2 joiners, and now this 4 transponders takes part into 4 Groups and each group participates to 1 joiner site. To be in force the constraint must these 4 groups to participate to the first 2 joiner sites, something like that, however is not possible.

Table 2: Notation for redeploy problem

$Group(X)$	The representation of one Group of Stream X
$Size_{old}(Group(X))$	The number of transponders for one group of Stream X, for previous tree representation
$Size_{new}(Group(X))$	The number of transponders for one group of Stream X, for current tree representation
$JoinerSites_{old}(Group(X))$	In how many sites takes part the one group of Stream X, for previous tree representation
$JoinerSites_{new}(Group(X))$	In how many sites takes part the one group of Stream X, for previous tree representation

6. IMPLEMENTATION

To evaluate our algorithm, we create from scratch a new system named ExaStream. ExaStream implemented into the Go programming language. The choice of the particular language, was because Go gives the appropriate mechanisms to write easy asynchronous procedures. ExaStream is similar to other systems [5] [1] [10] in the way that gives to the users the ability to define and execute their own topologys of user defined code. The decision to implement from scratch our own system is that we want to have the complete control of how the tuples are transferred to able to get clear conclusions.

Exastream is a system that executes jobs that can expressed as Directed Acyclic Graphs. The nodes in DAGs, define arbitrary user defined code and edges defines the way the data are combined. In figure 12 shows an ExaStream's DAG, that reads data from external sources and combines them using the DAB Join. The results of DAB Join passes to the User Defined function to process them further.

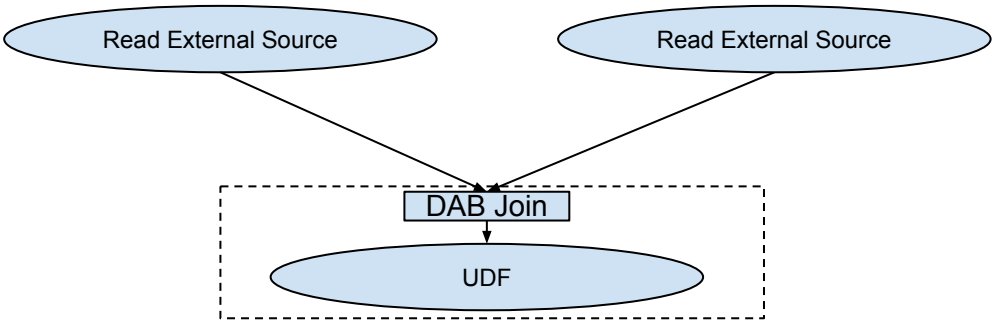

Figure 12: ExaStream DAG

ExaStream's DAGs can be defined from users using a programming API. For example the code in listing 2 shows how we can define a topology similar to the one that we described before. Specifically, this is the topology that we use to our evaluation. In the first line, we create the client responsible for executing the user defined computation DAG. The next two variables (rop1, rop2) initialized with the operators that produce the tuples. These operators, are user defined functions that produce tuples in a for loop with specific characteristics, as we will explain in Evaluation section. From line 17 to line 19 we construct our computation DAG. To add a computation node to our DAG we use the *AddComputationNode()* function. The first variable of this function„ defines the name of the node, the second the user define operator that this computation node will execute the third variable defines the way that the node will combine the input data, the fourth defines the input node names, and the last one the number of parallelism that we want to execute this UDF. The third variable of nodes ”Node1” and ”Node2” is the EmptyDT because these nodes do not accept any input. Node3 is the one that executes the DAB Join.

Exastream consists from three main components, workers, execution engine and clients. The workers are responsible to accept and execute new *tasks*. A task is the smallest unit of a job. An ExaStream task, determines the UDF that the worker must execute as

well as the tasks that this task will request to get data. Each task is the smallest unit of parallelism. In our example the ”Node1” operator will be partitioned in 16 tasks. The execution engine is responsible for accepting DAGs and transforming each DAG to a set of tasks. Afterwards, the execution engine sends these tasks to workers and informs the client when the execution has finished or when an error has occured. The client is the simplest component, its main work is to create and send DAGs to execution engines. Notice that Exastream can support more than one engine that manage the same resources in the cloud.

Listing 2: Code that defines an ExaStream’s computation DAG

```
1  dfl := client.NewDataFlow()
2
3  rop1 := exampleoperators.RangeOP{
4          NumberOfWindows: 200000,
5          RebalanceTriggerWindowNumber: 50000,
6          TuplesPerWindowAfterRebalance: 20,
7          TuplesPerWindowBeforeRebalance: 1,
8  }
9
10 rop2 := exampleoperators.RangeOP{
11         NumberOfWindows: 200000,
12         RebalanceTriggerWindowNumber: 50000,
13         TuplesPerWindowAfterRebalance: 1,
14         TuplesPerWindowBeforeRebalance: 1,
15 }
16
17 dfl.AddComputationNode("Node1", rop1, empty.NewEmptyDT(), []string{}, 16)
18 dfl.AddComputationNode("Node2", rop2, empty.NewEmptyDT(), []string{}, 16)
19 dfl.AddComputationNode(
20         "Node3",
21         exampleoperators.ReadOP{},
22         matrixjoin.NewDABJoinDT(1024, 10, 1),
23         []string{"Node1", "Node2"},
24         16)
25
26 dfl.Commit("127.0.0.1", "6999")
```

7. EVALUATION

In this section, we will describe the experimental evaluation of the DAB join. Our main goal, of the evaluation, is to show how the DAB Join behaves compared to the Symmetric Fragment and Replication Algorithm (Matrix Join). Our experiments aimed to explore how DAB Join scales changing the input rates of streams. Also in our evaluation, we take metrics about the time that joiners are blocked because they wait to finish a rebalance state and how the size of buffers and the number of cpus that DAB join uses affect these times.

7.1 Experimental Setup

We deployed our system to the Okeanos Cloud Infrastructure (www.okeanos.grnet.gr/) and used 8 virtual machines (VMs) each having a 2.100 GHz processor with two cores and 4 GB of main memory. We used synthetic streaming datasets. Specifically we produced tuples that have 5 columns. The first column was the timestamp and the others were dummy string values with size 8 characters.

To evaluate DAB joins, we produced finite number of data. More specific transponders produce 200,000 not overlapped windows in a for loop without latency. The ratio was 1 tuple per window and after the ⅓ of the total windows number the rate of one stream was changing. In our experiments we measured the execution time to finish this workload for DAB and Matrix join.

7.2 DAB Join vs Matrix Join

In this section, we will compare the Matrix Join with DAB Join. In diagram 13, we take the execution times when both algorithms uses 16 CPUs. As we mentioned, after the ⅓ of the total number of windows that are produced in our workload we change the rate of one stream. In our evaluation, the rate of one stream changed due to be 10, 20, 30, 40 times faster than the other.

Both algorithms, begin considering that the stream rates will be equals (mapping(4, 4)). As shown, DAB join adapts to changes of rates. This is the reason, why DAB Join is a lot faster than simple Matrix Join. Notice that, as the rate of one stream increases the more fast is the DAB Join compared to the Matrix Join. DAB join is slowest only when the rate doesn’t change. This happens, because we run the integer equation solver at the end of each window, something that Matrix Join does not need to solve.

The next diagram (diagram 14), shows how many tuples transferred from DAB Join and Matrix join as well the total number of tuples that are produced for the workload of previous experiment. As shown, the Matrix Join transfers 4 times more tuples than the produced ones. This happens because, Matrix Join use the mapping(4, 4) for all windows of workload. This means that Matrix Join replicates 4 times both streams. DAB join, on the other

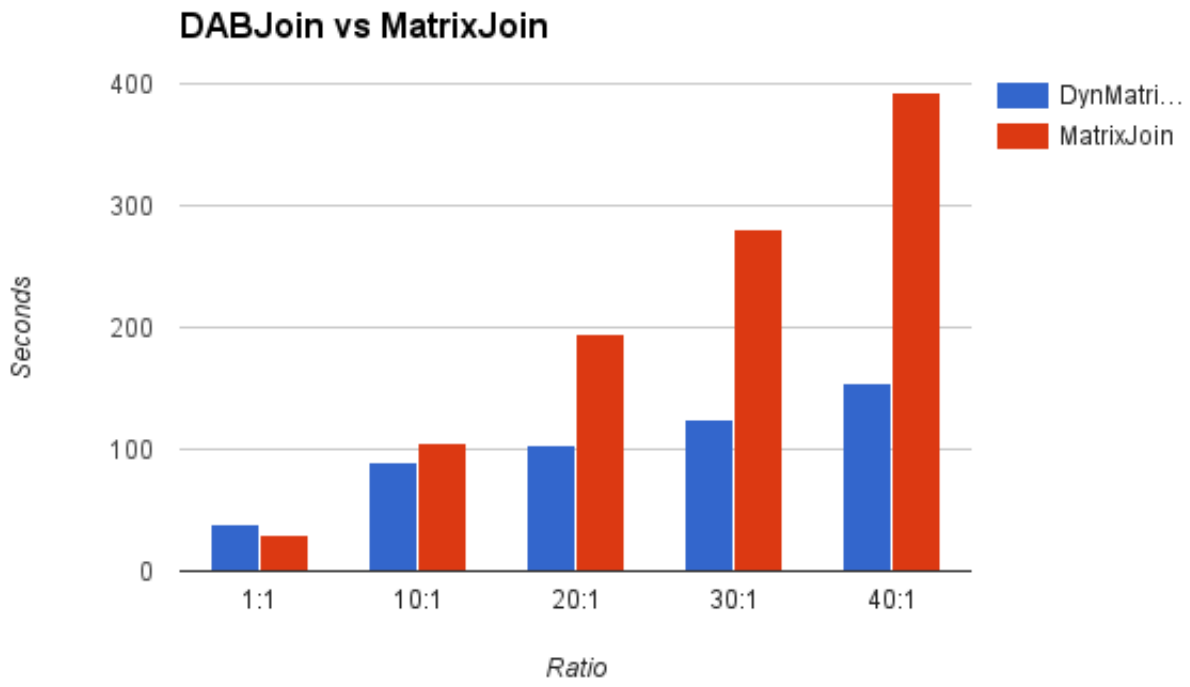

Figure 13: Evaluation of DAB Join in compare with Matrix Join when the input rates of streams changes

hand, after the ⅓ of the workload change the mapping to mapping(16, 1). This means that, replicates only one time the fastest stream and 16 times the slowest. For this reason, the Matrix join transfers much less tuples than the Matrix join and not much more than the total number of tuples that produced.

Notice that, the execution times with the number of tuples that are transferred follow the same path. This is reasonable because in our experimental setup we measure the times to transfer the tuples and we do not pay attention to the join algorithm executed in every joiner.

At the next experiment, (diagram 15) we evaluate how DAB Join and Matrix Join acts if we change the number of cpus that are available. In this experiment, the rate of one stream changes to 20 times faster than the other, after the ⅓ of the workload.

In our evaluation, we use as number of cpus perfect squares (4, 9, 16). This help Matrix join to choose the mapping that replicates equal both streams. As shown DAB join scales almost linear. This happens because, the tuples that produced from the stream with highest rate is much more for the other tuples that produced. So DAB join, replicated only one time this load and acts like a simple hash join.

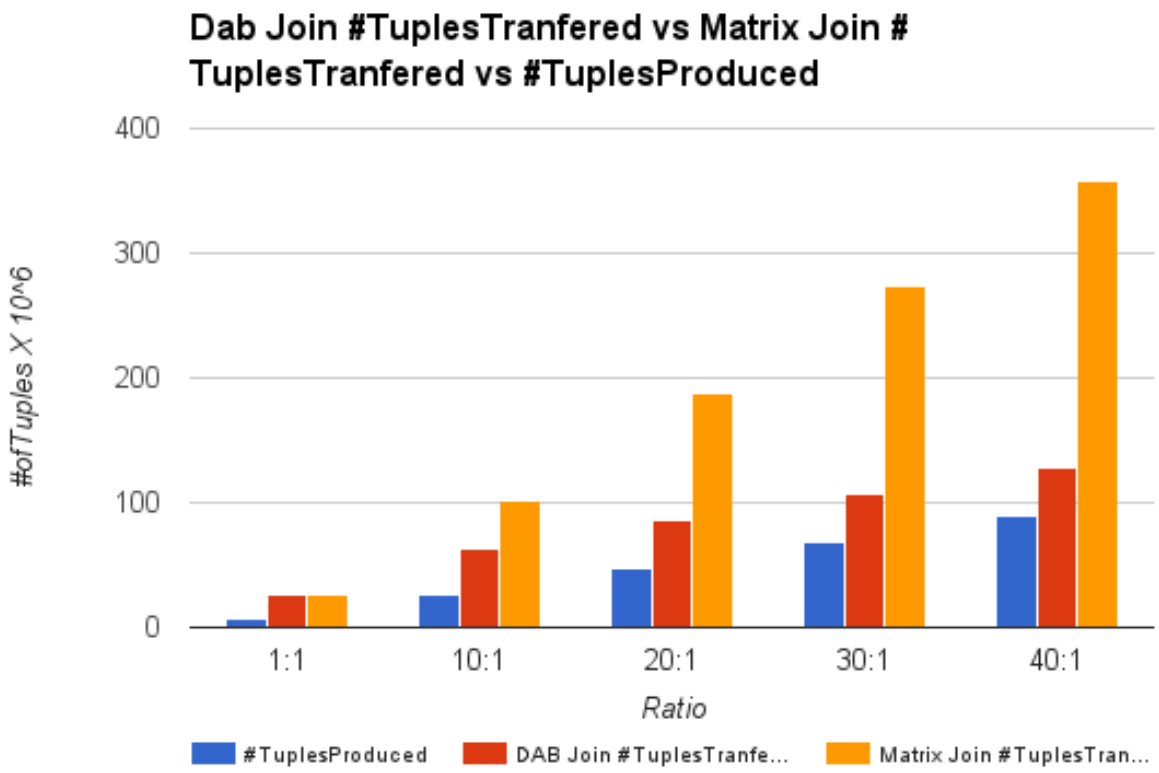

Figure 14: Evaluation of DAB Join in compare with Matrix Join when the input rates of streams changes

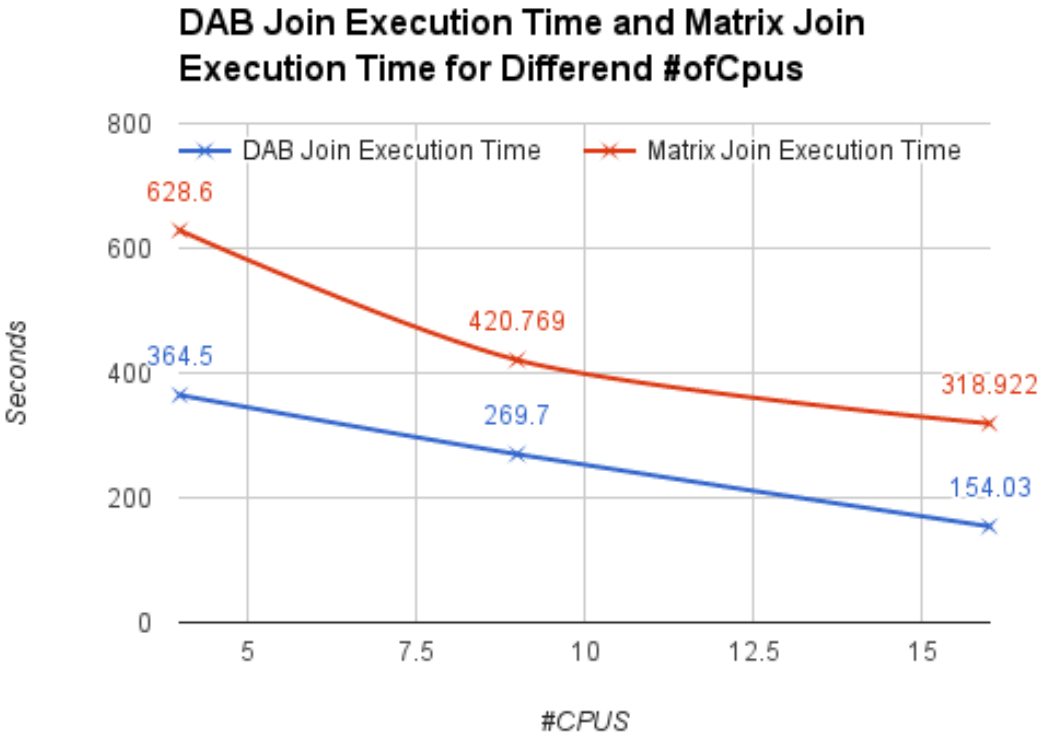

Figure 15: Evaluation of DAB Join in compare with Matrix Join when the input rates of streams changes

7.3 Rebalance Time

In this section, we will evaluate the time that joiners are blocked due to the rebalance phase. As shown, in the next experiments the factors that affect rebalance times is the number of CPUs and the size of buffers.

In diagram 16 we show how the number of cpus affects the rebalance times. As the number of cpus increases, the time of rebalance increases too. As we explain, this happens because a transponder needs to take a rebalance message that the field PreviousOutput is not empty. Until then, the transponder does not return a response for the other rebalance messages. So when we increase the CPUs we increase also and the number of operators. So the phenomenon of late response, is more often. In addition each transponder must process a lot more rebalance messages as the number of joiners increases.

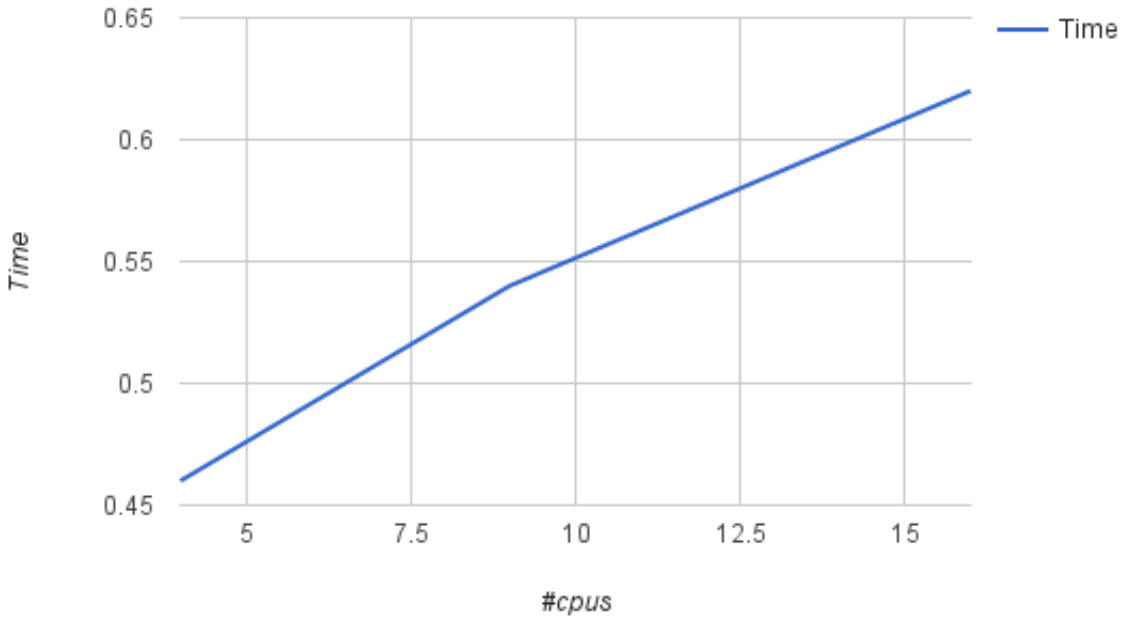

Figure 16: Evaluation of DAB Join in compare with Matrix Join when the input rates of streams changes

In the next experiment, we evaluate the rebalance time as the size of buffers increased while the number of cpus that DAB Join uses was 16. As we increase, the buffer size then the late response problem increases. This happens because some of the streams may stay behind. As we increase, the buffers it is more possible for a stream to stay behind. This is the reason that we take rebalance times that are more than 1.5 secs for very big buffer sizes (diagram 17).

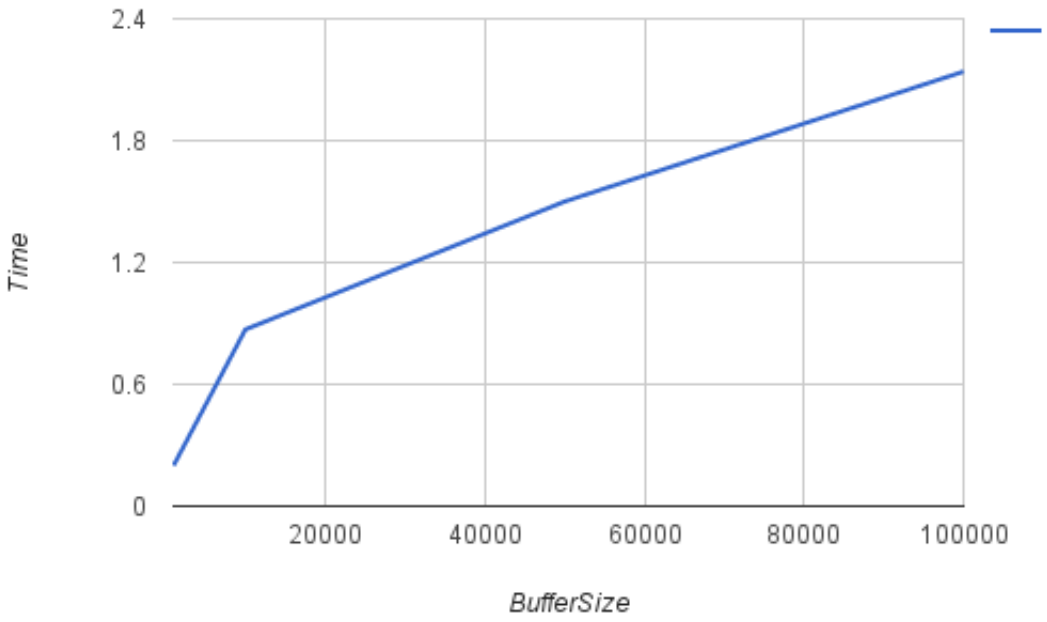

Figure 17: Evaluation of DAB Join in compare with Matrix Join when the input rates of streams changes

8. CONCLUSIONS

In this work we presented Distributed Adaptive & Balanced (DAB) Join. Dab join is a distributed n-way window join that adapts to the different input stream rates. DAB Join offers three characteristics that none of the existing distributed n-way window algorithm satisfies all-together: Specifically DAB Join:

- supports any kind of join predicates;
- minimizes the network cost while at the same time distributing the load equally to all nodes of the cluster;
- uses only one hop to distribute the data and execute the join, avoiding to distribute intermediate results that may be huge.

Our experimental results based on synthetic streams, show that our algorithm is scalable and adapts to changes on input stream rates. Also the experimental evaluation shows that the number of CPUs that a DAB join uses as well the buffer sizes affect the rebalance times. On our future work we intend to examine techniques for minizing rebalance times.

9. REFERENCES

[1] *Apache Flink*. https://flink.apache.org, 2016. [Online; accessed 19-July-2016].

[2] R. Avnur and J. Hellerstein, *Eddies: Continuously adaptive query processing*, Proc. of SIGMOD, (2000).

[3] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch, *Scalable and adaptive online joins*, VLDB, (2014), pp. 441–452.

[4] X. Gu, P. Yu, and H. Wang, *Adaptive load diffusion for multiway windowed stream joins*, ICDE, (2007), pp. 146–155.

[5] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kelogg, S. Mittal, J. Patel, K. Ramasamy, and S. Taneja, *Twitter heron: Stream processing at scale*, SIGMOD, (2015).

[6] O. Polychroniou, R. Sen, and K. Ross, *Track join: Distributed joins with minimal network traffic*, SIGMOD, (2014), pp. 441–452.

[7] W. Rodiger, S. Idicula, A. Kemper, and T. Neumann, *Flow-join: Adaptive skew handling for distributed joins over high-speed networks*.

[8] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin, *Flux: An adaptive partitioning operator for continous query systems*, ICDE, (2003), pp. 25–36.

[9] J. Stamos and H. Young, *A symmetric fragment and replicate algorithm for distributed joins*, Transactions on Parallel and Distributed Systems, (1993), pp. 1345–1354.

[10] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al., *Storm@ twitter*, in Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, 2014, pp. 147–156.

[11] S. Wang and E. Rundensteiner, *Scalable stream join processing with expensive predicates: Workload distribution and adaptation bytime-slicing*, EDBT, (2009), pp. 299–310.

[12] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar, *State-slice: New paradigm of multi-query optimization of window-based stream queries*, VLDB, (2006), pp. 619–630.