Scaling Out Schema-free Stream Joins

Damjan Gjurovski
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
gjurovski@cs.uni-kl.de

Sebastian Michel
TU Kaiserslautern (TUK)
Kaiserslautern, Germany
michel@cs.uni-kl.de

***Abstract*—In this work, we consider computing natural joins over massive streams of JSON documents that do not adhere to a specific schema. We first propose an efficient and scalable partitioning algorithm that uses the main principles of association analysis to identify patterns of co-occurrence of the attribute-value pairs within the documents. Data is then accordingly forwarded to compute nodes and locally joined using a novel FP-tree–based join algorithm. By compactly storing the documents and efficiently traversing the FP-tree structure, the proposed join algorithm can operate on large input sizes and provide results in real-time. We discuss data-dependent scalability limitations that are inherent to natural joins over schema-free data and show how to practically circumvent them by artificially expanding the space of possible attribute-value pairs. The proposed algorithms are realized in the Apache Storm stream processing framework. Through extensive experiments with real-world as well as synthetic data, we evaluate the proposed algorithms and show that they outperform competing approaches.**

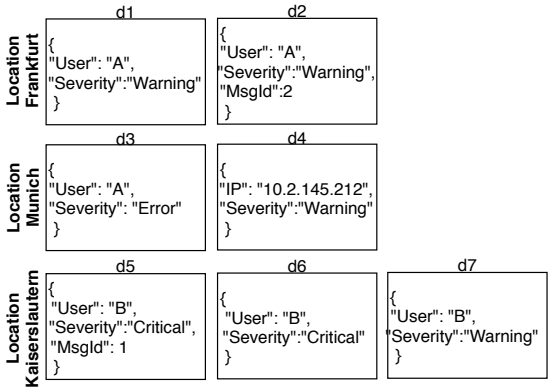


Fig. 1: Example for joinable documents

I. Introduction

The past years have witnessed a major shift from traditional data management over mostly relational data, toward various application-tailored data formats without fixed schema or even purely textual contents. Arguably, one of the most visible data formats is JSON [1], which eliminates the need to force data into relations, is designed to be human-readable, and has been accepted across various platforms and systems as the choice for data exchange. For instance, Twitter provides access to its public tweets via interfaces that deliver JSON data and platforms like geojson.io, likewise, enable uploading JSON data containing geo-coordinates for automatically placing them on a map. With the wide acceptance of JSON, not surprisingly, database systems tailored to handling JSON content, like MongoDB or CouchDB, are flourishing as part of the still ongoing NoSQL movement—while traditional vendors and communities around traditional RDBMS are extending their relational storage engines and query syntax to deal with JSON content, too. The flexibility JSON allows also comes with downsides when it comes to applying traditional query processing techniques to arbitrarily shaped data. In particular, the ubiquitously used relational join vastly benefits from fixed-schema relational data, to create appropriate hash tables or sorting orders, or using index structures across primary-foreign keys, for improved performances. In this work, we consider computing natural joins over schema-free JSON documents that arrive in a data stream. We will do so for a scale-out architecture, where the load of join computation is spread across multiple nodes that perform the actual join evaluation while being resource efficient and guaranteeing the exact join result.

Although performing joins in a distributed environment is a deeply investigated topic, to the best of our knowledge, most of the research is focused on performing traditional SQL-like equi or theta joins where the join parameters are either known in advance or provided at query time. We address the problem of computing *natural joins* in a distributed environment on top of schema-free JSON documents. If two documents share at least one attribute-value pair while at the same time they do not have any conflicting values for the same attributes, they will be provided as resulting joinable documents. For producing the correct join results, we rely on a partitioning strategy that ensures that potentially joinable documents will end up at the same machine, where they are subsequently joined. The final join result then enables the analysis of documents that contain complementary information, without knowing the join predicate in advance.

Consider the sample JSON documents displayed in Fig. 1, showing an excerpt of a company's server access log, with information about user logins and file accesses. Analyzing the stream of documents can provide beneficial insights for the company, such as identifying an attack on the servers, for instance, by detecting a constant failed login attempt on one location or identifying a user with constant failed access to files which may be a result of a virus-infected work station. The importance of the partitioning strategy is emphasized by the documents in Fig. 1. If the documents are partitioned based on their entire set of attribute-value pairs, the correct join

result will not be produced. Additionally, if they are partitioned based on an arbitrary attribute, the unequal occurrences of the attribute's values in the documents can contribute to creating partitions that differ greatly in the number of documents that they need to process.

A. Computational Model and Problem Statement

As the working model, we assume an incoming stream of documents D and a set of m machines over which the join computation is to be distributed. Each document consists of an unordered set of attribute-value pairs $d_i = \{a_1 : v_1, a_2 : v_2, \ldots\}$. Like for natural inner joins in traditional RDBMS, two documents are part of the join result, if and only if they have identical values for the attributes (columns) they share. In this paper, documents that do not share any attributes are excluded from the join result. We consider **window joins**, where two documents can only be joined if they belong to the same time (or count-based) window.

The task of data partitioning demands forming partitions that act as guideposts to lead incoming documents to the machine responsible for a partition. We form m partitions, one per machine. Partitions are expressed as sets of attribute-value pairs, too[1]. Depending on how the partitions are created, some documents might need to be sent to multiple machines, to ensure correct join results. We call this **replication** and aim at keeping it low. Additionally, we aim at fair **load balancing** across the compute nodes, which can be sometimes in conflict with low (or zero) replication. But one without the other goal can lead to impractical situations, like reaching zero replication by assigning all documents to a single machine or achieving perfect load balancing by sending all documents to all machines, to highlight the extreme cases. **The task of local join computation** needs to ensure that documents that arrived at one compute node are correctly and efficiently matched to documents that arrived earlier, then need to be stored (and indexed) to be matched to incoming documents next. Since we consider stream joins, a feasible approach has to adapt to changing data characteristics. That is, appearance of previously unseen attribute-value combinations or different co-occurrences between attribute-value pairs which have a negative impact on the partition quality (in terms of load balancing and replication).

B. Key Contributions and Outline

1) We propose a partitioning algorithm based on the concepts of association analysis. The created partitions lead to superior performance and scalability compared to existing approaches.
2) We discuss scalability limitations of partitioning approaches due to attributes with low value variety and how to circumvent this by deriving synthetic attribute-value pairs based on original ones.
3) We introduce a novel algorithm for computing natural joins over schema-free documents, based on frequent pattern (FP) trees.
4) We discuss a practical realization of our solutions in Apache Storm. The code is accessible through the personal websites of the authors.
5) We report on a detailed experimental study using real-world as well as synthetic datasets and compare our approach to competitors.

The remainder of this paper is organized as follows. Section II discusses related work. Section III sketches the high level scale-out architecture and gives a brief overview of Storm and its tuple routing primitives. The two main tasks, data partitioning and local join computation are presented in Section IV and Section V, respectively. Section VI discusses how the system reacts to documents that do not match any partition, how and when partitions are updated, and how to avoid limited scalability in presence of Boolean and other attributes of small value domain. Section VII reports on the results of the experimental evaluation, while, eventually, Section VIII concludes the paper.

II. Related Work

Join processing is a fundamental operation in relational database systems and further ubiquitous in many data management and analytic tasks, particularly also in data stream processing. There is, thus, a myriad of different approaches, ranging from traditional hash joins [2]–[4], variants of (index) nested loop joins, and sort-merge joins (various of them are now classical textbook knowledge), to more recent proposals of computing joins in scale-out architectures [11], [13], [14] or algorithms that exploit modern hardware [5], [6]. Joining data streams in a distributed environment is a topic that has been widely researched and most prominently visible in work which focuses on performing traditional SQL-like joins over tuples defined in a specific window. Kang et al. [7] analyze several stream join algorithms and inform the most suitable approaches for different scenarios. Recently the idea of a join-matrix model [8] has been revisited and reused for distributed join processing for both streaming [9] and map-reduce applications [10]. This model represents a join operation as a matrix, where dimensions correspond to the relations that need joining and the matrix cells represent the join output result. This approach does not scale well and suffers from a high memory consumption, due to the constant data replication. BiStream [11], which outperforms the join-matrix model, is a scalable and memory-efficient distributed stream join system. It is based on a novel join model, referred to as join biclique. Another distributed stateful stream joining system, called Photon [12], focuses on joining web search queries and clicks on advertisements in Google, by using a unique identifier present in both events. Unlike the MapReduce implementations of equi joins [13], [14], TimeStream [15] combines both MapReduce-style batch processing and streaming database systems. It exploits tuple dependencies to perform joins. The Pipelined State Partitioning (PSP) [16]

[1]Note that, in fact, it would also be possible to express partitions solely by attributes and not by attribute-value pairs. However, the restriction to attributes would not exhibit enough ways to (evenly) distribute computation to nodes, at least not for data or application cases we have observed.

approach focuses on distributed processing of generic multi-way joins with window constraints, by partitioning the states into disjoint slices and distributing the states using the created slices. Both PSP and TimeStream suffer from large communication overhead. Several algorithms also exist for a multi-core and main-memory environment [17]–[19]. ApproxJoin, introduced by Quoc et al. [20] is a distributed stream joining approach that provides approximate results. The approach is realized by using a Bloom Filter sketching and stratified sampling. D-Stream, introduced by Zaharia et al. [21], splits the streaming computation in deterministic batch computations and for each performs a MapReduce job. However, D-Stream is not applicable in our scenario since by grouping the data into small batches, candidate tuple pairs for joining may miss each other. Hence, this approach can only provide approximate join results, like ApproxJoin. Even though many of the approaches are scalable, efficient, and perform well for large datasets, our work majorly differs in the focus, which is performing stream joins when the join parameters are not given or known.

Another thoroughly analyzed research field, and a sub-problem of distributed stream joins, is data partitioning. A common observation is that approaches based on hash and range partitioning [22]–[25] suffer from poor load balancing in the presence of skewed distribution. Additionally, hash partitioning on several keys may result in scenarios where the joinable documents are not collocated on the same machine. The shuffle partitioning [24] blindly assigns tuples to machines, thus, it is inadequate for this approach since it will not place the same keys on the same machines. Since a document can be represented as a graph, graph partitioning methods are also applicable. The work presented by Alvanaki and Michel [26], continuously creates and maintains partitions of Twitter hashtags, to compute hashtag co-occurrences for trend analysis. We thoroughly analyze and compare with the proposed partitioning algorithms. The Kernighan-Lin algorithm [27] minimizes the edge-cut by moving vertices between partitions. The work presented by the authors of [27]–[29] is based on the Multilevel Graph Partitioning [30], where they improve the partitioning by combining it with different heuristics. In a dynamic environment, these approaches are computationally expensive due to the data changes as the stream flows, resulting in a partition that is valid only for a short time.

III. Architecture and Storm Preliminaries

Our goal is to assign partitions composed of attribute-value pairs to m machines. The incoming JSON documents are forwarded to the machines based on these partitions. As an example, consider the documents from Fig. 1 and assume that there are 2 machines. One possible partitioning could be:

- $pr_1 = \{"User":"A","Severity":"Warn.","Severity":"Err.","MsgId":2,"IP":"10.2.145.212"\}$
- $pr_2 = \{"User":"B","Severity":"Crit.","MsgId":1\}$

A document matches a partition if both share at least one attribute-value pair. This is important, as it reflects the requirement that two documents that even share just one attribute-value pair can potentially be joined. Based on the sample partitions, the documents d_1, d_2, d_3, d_4, d_7 will be assigned to the first machine and the documents d_5, d_6, d_7 will be assigned to the second machine.

The number of documents assigned to any of the machines represents the load. If we assume that the documents do not change over time, the machines responsible for pr_1 and pr_2 will have a load of 62.5% and 37.5% respectively. Ideally, all the machines should operate on an equal load. One can observe that the sum of the documents assigned to partitions pr_1 and pr_2 is larger than the actual number of documents. This is a result of replicating document d_7. This replication cannot be avoided since the document is joinable with both documents from pr_1 and pr_2. The number of replicated documents will represent the communication overhead because it represents the number of messages that need to be sent to every machine. Mutually exclusive partitions that have approximately equal load cannot always be achieved due to data characteristics. For that reason, the partitioning algorithm should aim to minimize the replication of documents for lower communication overhead and create partitions that distribute the load more equally among the machines. We will detail the proposed partitioning algorithm in Section IV.

Documents that are forwarded according to the partitioning are subsequently matched against previously arrived documents and stored for documents in the future—until the window dictates the eviction. To ensure efficient matchmaking, Section V introduces a novel join algorithm tailored to the peculiarities of schema-free data, using FP-trees.

A. Architecture

The architecture we realized is a rather general topology comprising multiple nodes within a compute cluster. We decided to use Apache Storm [31], [32], a mature and widely applied platform for streaming applications. Of central importance are those nodes that perform the actual join over the documents they get assigned. This assignment is done based on partitions, which are computed upfront and are continuously maintained. The task of computing partitions is parallelized, by creating a partitioning from disjoint samples of the data stream and, then, combining the partitioning into a final set of partitions. This resembles the generic topology presented by Alvanaki and Michel [26] for the task of counting hashtag co-occurrences. Fig. 2 depicts the topology. For every component, there can be either one (1) or multiple (n) instances or tasks and the lines depict the different subscription methods used for distributing the data among the instances of the components. We can observe the following components:

1) **Partition Creator**: creates partitions based on the attribute-value pairs of the currently seen documents. There can be multiple Partition Creators for sharing message load and computation.
2) **Merger**: receives partitions from the Partition Creators and merges them into a global set of partitions. It also updates the partitions. To satisfy the requirement of having a consistent set of partitions, only one instance of the Merger can be created.

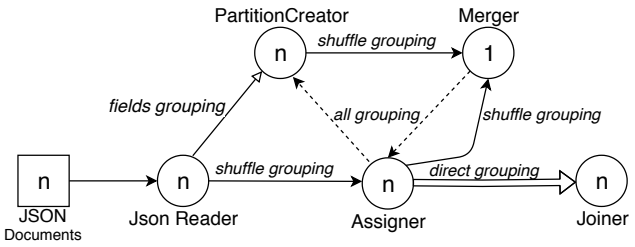


Fig. 2: Stream join topology

3) **Assigner**: simple dispatcher that forwards documents to the individual Joiner instances according to partitions. Informs the Merger once a document with unseen attribute-value pairs appears and initiates the recomputation of the partitions once the quality of the partitions is below a pre-defined threshold.
4) **Joiner**: responsible for computing the actual join between documents it got assigned.

B. Apache Storm

Apache Storm [31] is a distributed stream processing platform running parallel computations across a cluster of machines. Storm guarantees fault tolerance, guaranteed message delivery, and exactly-once tuple processing semantics. Applications realized in Apache Storm consist of components that are grouped into so-called Storm topologies. A storm topology can have two types of components, *Spouts* and *Bolts*, that work together to execute the logic of the application. The connection between the Spouts and Bolts is realized by using *stream groupings*. The Spouts represent the starting point of the topology, they are the source of the stream in the topology. The Bolts process the stream and optionally emit new streams from their computations. The parallelism in Storm is realized by specifying the number of instances for every component. This number is used by Storm for creating the required number of threads for every component. To perform the processing in parallel, the threads are split across the cluster of machines.

The data between the components in the topology is transferred in the form of tuples. Tuples represent a simple list of named values. For receiving data, Bolts need to register to the output stream of the appropriate Spout or Bolt. Storm provides different types of *stream groupings* that define how the tuples will be received by the multiple instances of the Bolt.

- **shuffle grouping**: randomly distribute tuples across the instances of the Bolt such that every instance will receive an equal number of tuples
- **fields grouping**: partition the stream based on one or more fields specified in the grouping
- **all grouping**: the stream is replicated across all of the instances of the Bolt
- **direct grouping**: the producer of the tuples decides which instance of the subscriber will receive the tuple based on a unique identifier

IV. Partitioning Based on Association Groups

Distributing the documents evenly among the machines with acceptable replication factor and load balance, urged us to develop a scalable and efficient partitioning approach. The proposed partitioning approach uses the attribute-value pairs of the documents to create the required partitions. By analyzing the patterns of occurrences of the attribute-value pairs, the documents are partitioned based on the meaning rather than their set of attribute-value pairs. In the following subsection, we present a partitioning algorithm based on *association rules*.

The algorithm is based on the observation that the occurrences of the elements within the documents are not arbitrary. The elements can depend on one another and regularly follow the same pattern of occurrence. Some elements constantly appear together, in groups, which we call *equivalence groups*. Additionally, some attribute-value pairs appear only when a specific attribute-value pair appears and not alone. We call the latter *association groups*. A clear connection to association analysis can be observed, but we do not adhere to the same principles for finding the association groups. Unlike the traditional association rule mining [33], we do not use support and confidence to eliminate groups that occur by chance. If the elements form a connection in at least one document, they will be considered as a valid association. The reasoning behind this choice is that we want our approach to provide results in real-time without having high computational complexity. For that purpose, we make a trade-off between the actual importance of an association group and the execution time of the algorithm. Detecting frequent itemsets is not only computationally expensive but will also lead to incorrect results. Considering only groups of elements with support and confidence above some predefined threshold will lead to having many documents that will not be assigned to any partition. As a result, they will not be emitted to any node and our statement of providing all documents that can be joined with one another will not be met.

The equivalence groups are identified by determining which attribute-value pairs satisfy the *equivalence relationship*. The association groups are calculated from the computed equivalence groups by establishing the equivalence groups that satisfy the *implies relationship*. We define the equivalence and implies relations as follows:

Definition 1: **Equivalence relationship** Let $D = \{d_1, ..., d_n\}$ be the set of all documents and $av_i = \{a_1 : v_1, ..., a_m : v_m\}$ be the set of attribute-value pairs for the document d_i. The attribute-value pairs $(a_i : v_i, a_j : v_j)$ form an equivalence group eg if whenever $a_i : v_i$ appears in document d_i then also $a_j : v_j$ appears and vice versa.

Definition 2: **Implies relationship** Let eg_i and eg_j be two computed equivalence groups. We say that eg_i implies eg_j if whenever the attribute-value pairs from eg_i appear in a document then also the attribute-value pairs from eg_j appear, but the attribute-value pairs of eg_j can appear individually without the attribute-value pairs of eg_i.

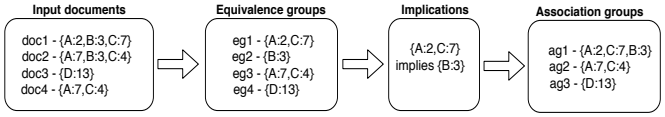


Fig. 3: Finding equivalence and association groups

Fig.3 represents an example of finding association groups. Based on the definition for the equivalence relationship, the algorithm determines that $eg_1 = \{A{:}2, C{:}7\}$ is an equivalence group since $A : 2$ and $C : 7$ always appear together in all the documents. Based on the equivalence groups it is determined that eg_1 implies eg_2 since in every document where $\{A{:}2, C{:}7\}$ appears also $B{:}3$ appears. By grouping together the equivalence groups that satisfy the *implies* relationship, the final association groups ag_1, ag_2, and ag_3 are produced.

A. Algorithm

The partitioning algorithm based on association groups is executed in two phases. In the first phase, the association groups are calculated from the attribute-value pairs of the documents and, in the second phase, the partitions are created based on the previously computed association groups.

The realization of the first phase is presented in Algorithm 1. The algorithm starts by creating a map referred to as $avInD$, where the key is a set of documents and the value is the attribute-value pairs that appear in that set of documents (Algorithm 1, Line 1). With the creation of the map, actually, the equivalence groups have been created since all the attribute-value pairs that appear in the same set of documents have been grouped together. The list EG is created by adding all the keys, the sets of documents, from the map $avInD$ (Algorithm 1, Line 2). The list of equivalence groups is sorted in ascending order based on the number of documents in every group (Algorithm 1, Line 3). The association groups are created by iterating over the list of equivalence groups EG and checking if the *implies relationship* is satisfied (Algorithm 1, Lines 5–15). If it is determined that equivalence group eg_i implies eg_j, then eg_j is removed from EG to avoid the creation of association groups with overlapping attribute-value pairs (Algorithm 1, Line 10). Additionally, for every association group ag_i, the load is computed by counting the number of documents in which the attribute-value pairs of ag_i appear (Algorithm 1, Line 13).

Algorithm 1 Association Groups–Based Algorithm

Require: Set of documents D
1: $avInD = getDocsForAvPair(D)$
2: $EG = avInD.keys$
3: sort(EG)
4: $AG = \{\}$
5: **for** i **in** $EG.size$ **do**
6: $\quad ag_i = EG[i]$
7: $\quad$ **for** $j = i + 1$ **in** $EG.size$ **do**
8: $\quad\quad$ **if** $EG[i]$ **implies** $EG[j]$ **then**
9: $\quad\quad\quad ag_i = ag_i \ \cup \ EG[j]$
10: $\quad\quad\quad EG = EG \setminus EG[j]$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad l_i = |\cup_i d_i|, \ ag_i \in d_i$
14: $\quad AG = AG \ \cup \ ag_i$
15: **end for**
16: **return** AG

The required m partitions are formed by using the nonoverlapping association groups generated by Algorithm 1. The initially empty partitions are populated with the first m association groups that have the highest load. Once the initial m partitions have been created, in every new iteration the association group with the highest load is selected and it is assigned to the partition that has the least load. This procedure is repeated until all the association groups have been assigned to partitions. Through this technique of assigning the association groups, partitions with approximately equal load will be created. The idea for assigning sets to partitions was already introduced by Alvanaki and Michel [26]. We can apply the same idea in our partitioning approach since both the association groups and disjoint sets partitioning algorithms as output create sets with nonoverlapping elements.

Applying this partitioning algorithm in our distributed Apache Storm topology requires one more step, the creation of the consolidated association groups. If the complete partitioning algorithm is used in every PartitionCreator, incorrect partitions will be created, as the PartitionCreators are oblivious of the whole set of documents on which the partitioning is performed. When they compute the association groups for their own set of documents, it does not mean that the same association groups will apply for the global set of documents. Thus, only the first phase of the partitioning algorithm (Algorithm 1) is executed by the PartitionCreators. The local association groups from every PartitionCreator will be emitted to the Merger that will have the responsibility to create the final, correct association groups and assign them to the m partitions.

To unify the association groups, the Merger first merges all the association groups that are a subset of another association group. For eliminating the duplicate attribute-value pairs, the Merger analyzes the association groups and if an attribute-value pair is part of two different association groups it is removed from the association group with more elements. Using the consolidated association groups the Merger populates the available m partitions following the approach explained above.

V. The FP-tree–Join for Local Join Computation

After the partitions are created and deployed in the system, potentially joinable documents are routed to the same compute node. The next step is to perform the actual join algorithm. For determining if two documents can be joined, we adhere to the join definition presented in Section I-A: two documents are joinable if and only if they have identical values for the attributes they have in common—documents that do not share any attribute cannot qualify for the join result. In the following,

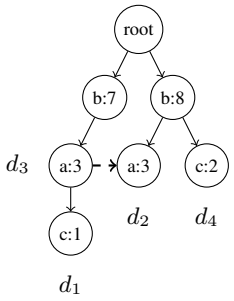


Fig. 4: FP-tree

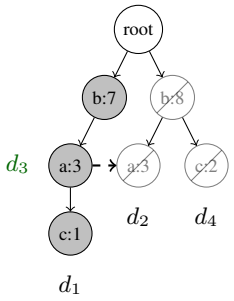


Fig. 5: FPTreeJoin (d_1)

we introduce a novel join algorithm that uses the FP-tree [34] as a foundation for discovering all the joinable documents.

A. FP-tree Creation

The FP-tree, introduced by Han et al. [34], is an extended prefix-tree that stores data in a compact way. Initially, the FP-tree was introduced in the field of frequent pattern mining, but with small modifications, it can be also applied for storing attribute-value pairs of documents.

For building an FP-tree, a strict ordering on the input elements must be imposed. We do this by sorting the attributes in descending order based on the number of documents in which they appear. If two attributes appear in the same number of documents, the attribute that has a smaller number of distinct values will have a higher priority in the ordering.

Table I contains an example document set. Following the approach for identifying the fixed ordering of the attributes, attribute b will have the highest priority in the ordering since it appears in most documents. The attribute a appears in more documents than the attribute c which results in the final fixed ordering $b \rightarrow a \rightarrow c$. For convenience, the reordered documents according to the aforementioned attribute ordering are displayed in the rightmost column of Table I.

Initially, the FP-tree is empty, consisting of only the root node labeled with 'null'. Every node in the FP-tree will consist of a label, which in our context will be an attribute-value pair and a pointer to the next node in the FP-tree that has the same label. There is a list in every node that is used for storing the document ids from where the attribute-value pairs of the branch have been inserted. The list of document ids will be used for identifying the joinable documents. As in the original paper [34], the same usage of the header table is retained, i.e., connecting nodes with equal labels. For every branch of the FP-tree, a unique branchId will be created. In Fig. 4, the FP-tree for the documents in Table I is shown.

TABLE I: Four sample documents and their reordered representation imposed by the global attribute order (right column)

ID	Attribute-Value Pairs	Ordered AVP
d_1	$\{a{:}3, b{:}7, c{:}1\}$	$\{b{:}7, a{:}3, c{:}1\}$
d_2	$\{a{:}3, b{:}8\}$	$\{b{:}8, a{:}3\}$
d_3	$\{a{:}3, b{:}7\}$	$\{b{:}7, a{:}3\}$
d_4	$\{b{:}8, c{:}2\}$	$\{b{:}8, c{:}2\}$

To use FP-trees at the join processing nodes in our topology, we first have to compute an attribute ordering, which is done right after the partitions are created and the documents start getting assigned to the join processing nodes. The documents received at the *Joiners* are then tested against the stored documents and then added to the tree, to be matched with forthcoming documents. This means that the creation of the tree and the matchmaking is done in a batch fashion. In our current version, we investigate so called tumbling windows, which define nonoverlapping chunks of documents. This allows to evict the entire tree once the window tumbles. For sliding windows, tree updates or frequent tree evictions and rebuilds are required, which we believe is beyond the scope of this paper and part of our ongoing work.

B. FPTreeJoin Algorithm

For producing the join partners while efficiently traversing the constructed FP-tree, we introduce the *FPTreeJoin* algorithm. The FPTreeJoin algorithm is tailored for deep trees that are a result of having documents where the most frequent attribute has a small number of distinct values. This algorithm is based on the observation that if there is an attribute that is present in all the documents, we can immediately ignore large portions of the tree. This observation is easily explainable if there is a Boolean parameter present in every document. Because this parameter appears in the highest number of documents and at the same time it has the smallest number of distinct values we can be certain that it will have the highest priority in the fixed ordering. As a result, the nodes labeled with $bool : true$ and $bool : false$ will be immediate children of the root. By traversing the tree directly through the children of the root we can instantly prune half of the tree and reduce the execution time complexity of the algorithm. We generalize this conclusion and we consider not only the children of the root but all parameters whose attributes appear in all the documents. If there are n parameters of this kind, we can be certain that the first n levels of the FP-tree will consist of only these parameters. This means that instead of navigating through the nodes of these n levels and searching for nodes whose attribute-value pairs are not in conflict, the algorithm can directly obtain the equally labeled node from the level and ignore all the other nodes on that same level.

Algorithm 2 FPTreeJoin

Require: Document d_i, Root of FP-tree $node$, Number of attributes that appear in all documents num

1: $result = [\ ]$ //Final list of joinable documents
2: **while** $j < num$ **do**
3: $\quad child = root.children.get(d_i.avPairs[j])$
4: $\quad node = child$
5: $\quad result.add(node.documents)$
6: **end while**
7:
8: $result.add($**traverse**$(node, d_i.avPairs, result)$
9: **return** $result$

Algorithm 3 FPTreeJoin - Traversal of Children

Require: Node $node$ from the FP-tree, Set of attribute-value pairs avP, List of joinable documents $result$

```
1: function TRAVERSE
2:     result.add(node.documents)
3:     for child in node.children do
4:         cAvP = child.avPair
5:         if cAvP.attr ⊂ avP.attrs and cAvP ⊄ avP then
6:             continue
7:         end if
8:         traverse(child, avP, result)
9:     end for
10:    return result
11: end function
```

Once the efficient navigation through the first n levels is accomplished, and there are no more attributes that appear in all the documents, depth-first search navigation through the remaining levels is employed.

The structure of the FPTreeJoin algorithm is presented in Algorithm 2. Through the creation of the FP-tree, we can obtain information about the number of attributes that are present in all the documents and pass this information through the variable num to the algorithm. This is used in order to quickly navigate through the first levels of the FP-tree consisting of the ubiquitous attributes (Algorithm 2, Lines 2–6). All documents that have been encountered along the path are stored in the $result$ list as resulting joinable documents for document d_i. Once a large portion of the FP-tree is pruned and there are no more attributes that are present everywhere, the method $traverse()$ is called (Algorithm 2, Line 8). Using the $traverse$ method the algorithm will continue to navigate toward the leaf nodes of the tree and gather all the documents that can be joined with the analyzed document.

Algorithm 3 provides an overview of the $traverse$ method used for traversing the remaining children of the tree. The algorithm navigates through the branch provided as input in a top-down fashion deciding whether the documents, represented by the nodes of the branch, are join partners for the document provided as input. Document ids stored in the nodes for which the join test is satisfied are collected (Algorithm 3, Line 2) and the algorithm continues to navigate down through the branch (Algorithm 3, Lines 3–9). If the join test is unsatisfied, the algorithm ignores the current node and all of its successive children (Algorithm 3, Lines 5–7). Once a leaf node is reached, the $result$ is returned (Algorithm 3, Line 10).

An important remark for the traverse method (Algorithm 3) is that not all document ids from the nodes can be added as joinable documents. If there are no ubiquitous attributes in the documents, the loop in Algorithm 2 will not be executed. As a result, the currently investigated document will not have any in common attribute-value pairs with the branch of the node *node*. For that reason, it is important to keep track of all the shared attribute-value pairs of the investigated branch with the document d_i and update them for every new child node. A document id can be stored as a joinable document only if the number of shared attribute-value pairs is greater than 0.

In Fig. 5 an example for finding the joinable documents for document d_1 is shown. The algorithm receives as input that there is only one level in the FP-tree that has an attribute that appears in all the documents, i.e., the first level. Thus, $b:7$, the first attribute-value pair of the document d_1, is used for selecting the equally labeled node from the first level of the FP-tree. Consequently, the branch of the node $b:8$ will be pruned and not considered in further steps. The $traverse$ method is performed for the node $b:7$ which identifies that only document d_3 can be joined with document d_1.

VI. Practical Tweaks: Handling Dynamics and Countering Low Value Variety

A. Updating and Recreating Partitions

Over time, new documents with previously unseen attribute-value pairs will arrive in the system. It is the responsibility of the Assigners to handle these documents since they represent the components that connect the partitions with the Joiners.

We define the updating of the partitions as adding a single document to the already created partitions. The updating is triggered by the Assigners and it is performed in the Merger. Because of the strict requirement of having only one Merger, updating the partitions for every document with previously unseen attribute-value pairs is impractical, for two reasons. Firstly, the Merger will be congested with many messages, which will directly affect the performance of the whole system. Secondly, we risk incorporating extremely rare occurrences into the partitions that spoil the quality of the partitions in terms of replication. Thus, a document is a candidate for an update if it has an attribute-value pair that has appeared at least δ times. We consider that the attribute-value pairs that appear less than δ times are a result of a unique event and as such, they should not be considered for updating the partitions. However, we guarantee to produce all the joinable documents for a given window. Hence, documents that do not satisfy the update requirement are emitted to all Joiners.

As indicators for triggering the repartitioning, we consider the load balance and the replication. With the creation of the final partitions, the Merger computes the load balance and replication of documents that are a direct result of the computed partitions. When the Assigners detect that either the load balance or replication factor has increased over a prespecified threshold θ, they inform the Partition Creators and the Merger that in the next window recomputation of the partitions should be initiated.

B. Introducing Value Variety for Scaling

Attributes that have only a few unique values and are present in all of the documents will limit the scalability of the partitioning approach, as there will not be enough partitions created. As an extreme but not unrealistic case, consider documents containing a Boolean parameter, where by intuition, only two partitions should be created. In order to scale the computation to more than just two machines,

there needs to be more variety in the domain of attribute values. The approach we conduct first searches for an attribute, referred to as *disabling attribute*, which appears in **all** the documents and has a number of unique values that is smaller than the required number of partitions. To enable the creation of the required number of partitions m without introducing unnecessary replication, the disabling attribute's values will be "concatenated" with the values of the next attribute that appears in **most** of the documents and has the smallest number of unique values, called *combining attribute*. This expansion is repeated until the artificially created attribute has enough distinct values to satisfy the required number of partitions m.

As shown in Section VII, this expansion of attribute values is also a necessity for the disjoint sets algorithm [26], because, without it, it can practically never create enough partitions to populate the available number of machines.

Note that this expansion approach can introduce additional replication of documents since there can be documents in which the combining attributes do not appear—and hence, the artificial value cannot be created at all. As a consequence, to guarantee correct join results, such documents will be emitted to all machines, thus, creating additional network load (replication). The expected replication can be estimated by $p_{na} * m$, where p_{na} represents the percentage of documents where the parameter av_i does *not* appear and m is the number of partitions (i.e., machines).

VII. Experiments

We have implemented our proposed topology using Java 8 and Apache Storm 1.2.2. For carrying out the experiments, the topology was deployed on a cluster of 8 servers, with Linux (Ubuntu 14.04.5) as operating system. Every machine in the cluster is equipped with two 6-core Intel Xeon E5-2603 v4 CPUs @1.7 GHz and 128GB RAM.

A. Competitors

We investigate the performance of the following partitioning algorithms.

- Our Association Groups based partitioning approach (**AG**)
- The approach based on set covers (**SC**) [26], tuned for low communication overhead. The SC algorithm is based on the set cover problem and tries to find the smallest set of subsets through which the occurring sets can be represented. In this work, the attribute-value pairs of one document represent a set. It starts by creating initial partitions where in every iteration the set with most uncovered attribute-value pairs and minimal number of covered attribute-value pairs is selected. For assigning the remaining sets to the initial partitions, in every iteration, the set with least number and most uncovered attribute-value pairs is selected and assigned to the partition that has the least load and the most attribute-value pairs in common with the selected set.
- The disjoint sets (**DS**) partitioning approach [26]. The DS algorithm creates connected components called disjoint sets by combining all the sets that share at least one attribute-value pair. Every attribute-value pair belongs to one and only one disjoint set. The partitions are created by assigning every disjoint set to exactly one partition. By creating nonoverlapping partitions, the DS algorithm achieves perfect replication of having every document in exactly one partition.

For the FP-tree–based join algorithm (**FPJ**) we conduct experiments with two algorithms, the Nested Loop Join (**NLJ**) and Hash–Based Join (**HBJ**). The HBJ creates a hash table on the individual attribute-pairs as keys, such that documents that overlap in at least one attribute-value pair can be determined efficiently, essentially resulting in some sort of inverted index over the contents of the documents.

B. Datasets

The experiments were carried out using two different datasets. The real-world dataset (**rwData**) that consists of server logs of a mid-size company. The server logs, gathered from 5 servers, provide information about the user logins and file accesses. The data consists of *46 million* JSON documents collected in a period of *105 days*. To simulate a streaming environment, we take the daily produced amount as the number of documents produced every 3 minutes. We consider the whole data as a stream of incoming tuples, where the difference in the fields of the server logs fulfills our schema-free requirement for the documents. The sample in Fig. 1 gives an impression of how the data looks.

For the synthetic datasets (**nbData**), we employ the NoBench JSON data generator proposed by Chasseur et al. [35]. The NoBench data generator creates an array of JSON objects with several attributes and in our work, every JSON object represents a separate JSON document. To create the possibility of having joinable documents, we remove the attribute num from the data generator because this attribute is unique in all the objects that are produced. It is important to note that the NoBench dataset consists of largely diverse elements, resulting in situations where every successive window contains documents with many previously absent attributes. Surprisingly, the same phenomenon holds for the real-world dataset. By investigating several window sizes, we deduced that in every subsequent window large number of the documents consist of previously unseen, new attribute-value pairs. This will be clearly visible in our experiments.

C. Performance Metrics

During the execution of the aforementioned algorithms, we gather the following performance measurements.

- **Replication:** We measure the replication for every window as the average number of times that the same document has been sent from the Assigners to the Joiners. We also consider the documents that have unseen attribute-value pairs that do not satisfy the requirement for updating the partitions.
- **Maximal Processing Load:** We define the processing load of a single Joiner as the percentage of the assigned

number of documents out of the total number of documents emitted for a given time window. The maximal processing load represents the highest processing load at one of the Joiners.

- **Load Balance:** To assess the quality of the partitioning in terms of balanced load across the compute nodes, we measure the Gini coefficient[2]. It represents how much the actual load distribution deviates from a perfectly equal load distribution.

D. Configuration Parameters

For testing the performance of our topology several settings for different parameters were examined.

Number of partitions m: the number of partitions is directly connected to the number of Joiners. There will be as many Joiner instances as the number of partitions. By increasing the number of partitions, the load for every instance of the Joiner is minimized but on the other hand, the replication can be increased by spreading the same documents on different machines. In our experiments, we performed measurements for $m = 5$, $m = 8$, $m = 10$, and $m = 20$.

Window size w: the size of the window specifies the number of documents that will be used in the Partition Creators for creating the initial partitions. Through the size of the window, the replication is directly affected. If the window is too small, the created partitions are not representing the data well, which leads to many documents having attributes that are not present in any of the m partitions. For our experiments, the size of the window was set to 3, 6, and 9 minutes.

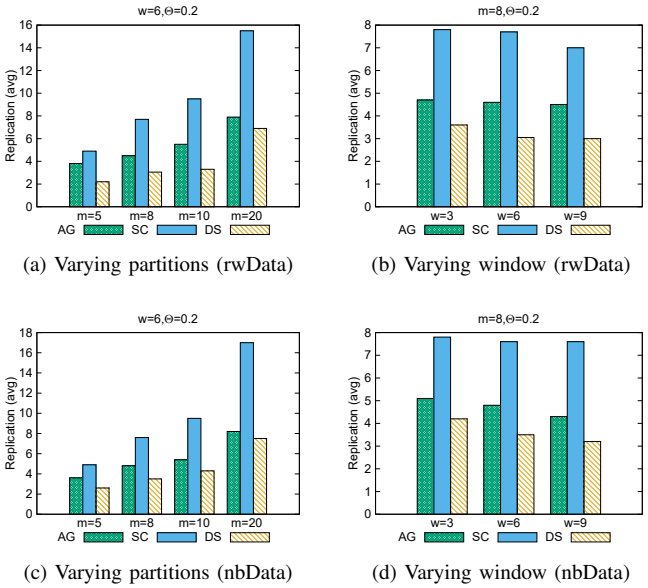
Repartitioning threshold θ: the repartitioning threshold is a boundary that defines when a recomputation of the partitions should be initiated, as described in Section VI-A. When the replication or processing load has changed for more than the repartitioning threshold, the Assigners inform the Partition Creators that in the next window a recalculation of the partitions should be performed. A smaller threshold indicates more frequent computation of partitions whereas a larger value indicates higher network traffic and a higher load per machine. We set the repartitioning threshold to 0.2 and 0.6, where a value of 0.2 means that the replication or processing load has increased by more than 20%.

E. Experimental Results

We investigate the effects of the parameters on the approach by varying one parameter while keeping the other fixed. If not specified otherwise, we use the default values $m = 8, \theta = 0.2$, and $w = 6$. All settings use six Assigners.

If a previously unseen attribute-value pair arrives at the Assigner at least $\delta = 3$ times (cf., Section VI-A), the partition recreation is initiated. The statistics for computing the quality of the partitions are performed at the end of every window. The Joiners compute and report the join results for every window. For the NoBench dataset, all partitioning approaches use the attribute expansion technique, presented in Section VI-B, as there is a Boolean attribute. Without the attribute expansion, the approaches will not scale to a larger number of partitions/machines. The disjoint sets algorithm (DS) still needs the expansion of attributes for the real-world data, because, no matter the size of the window, it constantly creates disjoint sets whose number is smaller than the number of partitions we aim at.

[2]http://en.wikipedia.org/wiki/Gini_coefficient



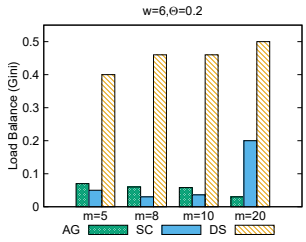
(a) Varying partitions (rwData) (b) Varying window (rwData)

(c) Varying partitions (nbData) (d) Varying window (nbData)

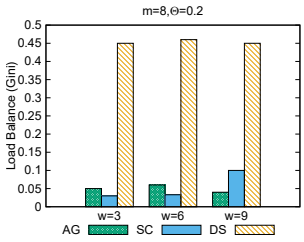
Fig. 6: Replication

1) Replication: The changes in the replication concerning the different settings of the configuration parameters are presented in Fig. 6. One can observe that the DS algorithm does not have the expected (perfect) replication of 1 (a document belongs to only one machine). This is a result of documents that do not match the existing partitions and, as such, are emitted to all machines. This fact contributes to the higher replication of the other algorithms, too. In Fig. 6a and Fig. 6c we can observe that the number of partitions has a large impact on the replication. The AG approach handles the increase in the number of partitions acceptably well, by creating partitions without many overlapping documents. Additionally, as the number of partitions increases, the comparison of the replication with respect to the worst-case scenario improves, indicating the scalability of our partitioning approach. On the other hand, the SC approach, no matter the number of partitions, always approaches the worst possible replication of sending every document to (almost) every machine. By increasing the window size, the AG and DS algorithms create more suitable partitions. Based on the measurements in Fig. 6b and 6d, we see that this is not the case for the SC approach. The SC algorithm, most of the time creates partitions where it needs to emit every document to almost all available machines.

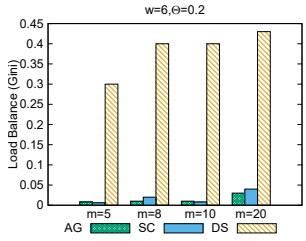
For a distributed system the replication factor is of great importance, as it is directly related to the network traffic and also to the load the individual worker nodes have to cope with.



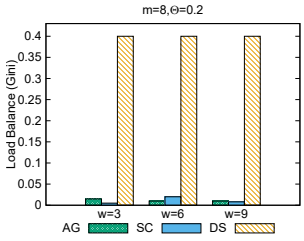
(a) Varying partitions (rwData)



(b) Varying window (rwData)

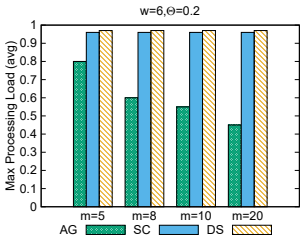


(c) Varying partitions (nbData)

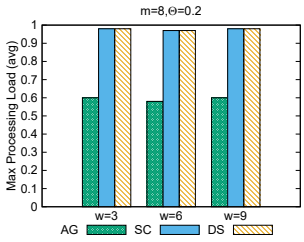


(d) Varying window (nbData)

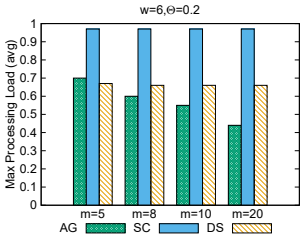
Fig. 7: Load balance



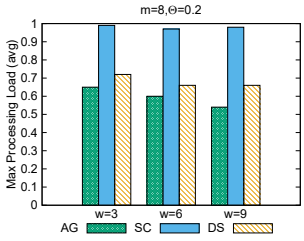
(a) Varying partitions (rwData)



(b) Varying window (rwData)



(c) Varying partitions (nbData)



(d) Varying window (nbData)

Fig. 8: Maximal processing load

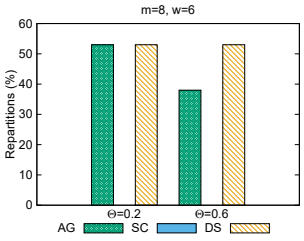
From the measurements presented in Fig. 6, we can conclude that the DS algorithm has the best replication followed very closely by our AG approach. The SC approach is completely not suitable for our documents and moreover, it only creates unnecessary overhead since the same results can be achieved by emitting every document to every machine. However, the large downside of the SC approach becomes evident, once we inspect the load balance.

2) Load Balance and Maximal Processing Load: We use the measurements for the load balance and the maximal processing load to show how the machines are balanced in terms of load and whether the load balance is a result of an appropriate partitioning strategy or a result of replicating the same documents across the different machines.

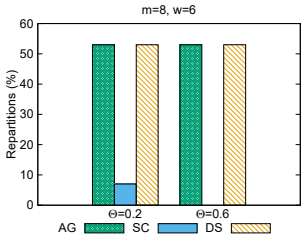
Fig. 7 reports the load balance when changing the configuration parameters. As it can be observed, both the AG and SC approaches have satisfactory load balance, which for the synthetic dataset is not affected by varying the number of partitions or window sizes. For the real-world dataset, the load balance of the AG approach improves by increasing the number of partitions. This is a result of how the association groups are assigned to the partitions. By having more available partitions, the AG approach distributes the association groups more evenly across the partitions, creating a more balanced distribution of the documents. Unlike the AG and SC approach, the DS approach provides an inadequate distribution of the documents among the machines. This indicates that the DS algorithm creates disjoint sets which are extremely different in the number of documents that they have.

Based on the measurements for the processing load, we can conclude that both the AG and the SC approach are a good choice for partitioning schema-free documents. By examining Fig. 8 we can see that this is not the case. The SC partitioning approach in every setting has at least one machine where almost the complete set of documents has been assigned. Thus, the balanced load is not a result of a good partitioning approach but it is achieved by replicating almost all the documents on the available machines. On the other hand, the AG algorithm has a lower maximal processing load when increasing the number of partitions and the window size. This leads to the conclusion that the AG partitioning approach is highly scalable which is not a result of replicating more documents but of efficient computation of the partitions. Similar to the SC algorithm, the DS algorithm does not improve in terms of the maximal load when increasing the number of partitions. Moreover, for the real-world data, there is a single machine that receives almost all documents. This further underpins the expectation that the DS approach is not appropriate for larger JSON documents where a substantial number of documents are connected to one another via one or more attribute-value pairs.

3) Repartitions: The repartitioning is triggered either as a result of increased replication or unbalanced load. In Fig. 9 we

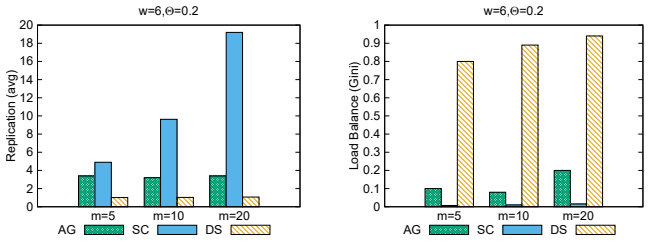


(a) Varying threshold (rwData)

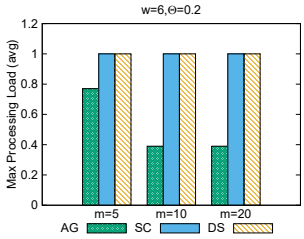


(b) Varying threshold (nbData)

Fig. 9: Number of repartitions

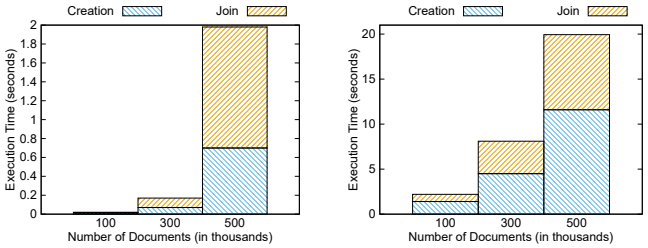


(a) Replication (b) Load balance

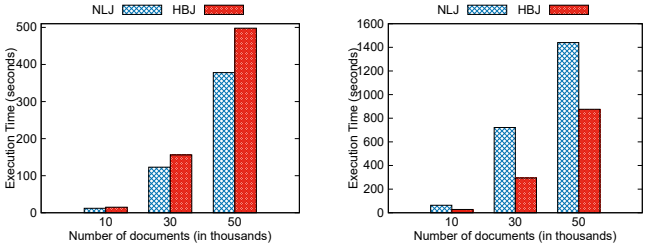


(c) Max processing load

Fig. 10: Ideal execution



(a) FPTreeJoin (rwData) (b) FPTreeJoin (nbData)



(c) Competitors approaches (rwData) (d) Competitors approaches (nbData)

Fig. 11: Execution time comparison

present the number of times the repartitioning was performed as a percentage of the total number of executions. The AG partitioning algorithm performs as expected for the real-world data, it has lower replication as the repartitioning threshold increases. For the NoBench data, the AG approach has the same replication rate for the different thresholds, 50% of the total number of elapsed windows. This is a result of the magnitude of unseen documents that arrive in every subsequent window and cause the algorithm to compute the partitions in every second window. The DS algorithm constantly has the same repartitioning rate. This comes as no surprise since having a previously unseen portion of documents in the subsequent window produces increased replication that undoubtedly will be higher than the originally computed average replication with value 1. The SC algorithm, from the first window, computes the worst possible partitions by replicating every document on almost every machine and from that point, there is no load balance or replication that is inadequate enough to cause the repeated computation of the partitions.

4) Ideal Execution: The presented results for replication, load balance, and maximal processing load are directly affected by the previously unseen attribute-value pairs that cause the containing document to be emitted to all the machines. Thus, the new attribute-value pairs directly increase the processing load and replication of documents. To demonstrate the execution of our approach when the data characteristics (in terms of co-occurring attribute-value pairs) are largely stable, we generated a dataset derived from the real-world data where we take one time-window and we repeat it multiple times. In every new window, we only add a predefined, small number of previously unseen documents. The results for the measurement are presented in Fig. 10. Immediately noticeable is the improvement in the replication of the AG approach compared to the general case. Having a small number of previously unseen attribute values pairs indicates that the measured replication is a direct result of the partitioning algorithm. By increasing the number of available partitions, the AG approach continuously produces partitions with acceptable replication and improves the maximal processing load which contributes to better communication overhead and a more balanced load. The conclusions that we have made for the partitioning approaches are confirmed and even more evident. The results for the measuring of the maximal processing load show the scalability and efficiency of our approach and they prove that the excellent replication factor and load balance are indeed a consequence of a superior partitioning approach.

5) Local Join Execution Time: To the best of our knowledge, the existing approaches for joining documents in a streaming environment focus on performing equi or theta joins (or both) where the join attributes are known in advance or are of no concern. In our work, the attributes on which the join will be performed are unknown, but still, we want to exploit their importance when deciding if two documents are part of the natural join result. We compare our join approach with the natural join approaches used in traditional database systems, the Nested Loop Join (NLJ) and the Hash-Based Join (HBJ). The FPTreeJoin algorithm, as well as the baseline algorithms, are not distributed algorithms and as such are performed entirely within the Joiner instances. The execution time measurements of the join approaches for the real-world and synthetic datasets are shown in Fig. 11. The x-axis in Fig. 11 informs about the number of documents used for performing the join algorithm on a single node. The presented measurements depict the execution time of the join algorithms over a fixed number of documents on a single compute node and not a complete window in the streaming application. In

our proposed topology, the join is performed for every window where every Joiner operates on the assigned documents.

Evidently, our proposed join algorithm is superior when performing joins over large, schema-free JSON documents. The time needed for both creating the FP-tree and performing the FPTreeJoin approach when handling ten times more documents is orders of magnitude better. Furthermore, the execution time of the join algorithm is not significantly impacted by the data size. Even for the more diverse documents produced with the NoBench generator, the join is computed in a matter of seconds. One can immediately observe that the execution times of HBJ and NLJ become unacceptably high for real-time analysis. For joining 50,000 documents, considered a small amount in a streaming environment, both approaches take several minutes. This becomes even more evident for the synthetic dataset where the NLJ algorithm needs more than 20 minutes and the HBJ algorithm around 15 minutes. One more interesting phenomenon that can be observed is that both algorithms perform differently for different datasets. For the real-world data, the NLJ algorithm outperforms the HBJ algorithm. This is a consequence of the high interconnection between documents, resulting in large document lists for a single hash value. For the synthetic dataset, the unique attribute-value pairs contribute to smaller bucket lists that drastically improve the execution time of HBJ.

VIII. Conclusion

We addressed the problem of computing natural joins over schema-free JSON documents, which entails two core tasks: finding a feasible data partitioning schema and performing local query evaluation efficiently. For both aspects, we put forward novel solutions, harnessing core concepts of association analysis for partition creation and an FP-tree–based join algorithm for local execution at the involved compute nodes. We further provided a countermeasure for cases of low attribute-value variety that inherently limit scalability. We reported on results and insights of a performance evaluation, which clearly showed the viability of the overall approach to handle large volumes of data in a resource-efficient manner. Further, performance gains of the partition approach over existing competitors and the vast superiority of the FP-tree–based join over baseline approaches were observed.

References

[1] "JavaScript Object Notation." https://www.json.org/.

[2] R. Barber, G. M. Lohman, I. Pandis, V. Raman, R. Sidle, G. K. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe, "Memory-efficient hash joins," *PVLDB*, vol. 8, no. 4, pp. 353–364, 2014.

[3] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Comput.*, vol. 1, no. 1, pp. 63–74, 1983.

[4] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," *SIGMOD*, pp. 1–8, 1984.

[5] S. Manegold, P. A. Boncz, and M. L. Kersten, "Optimizing main-memory join on modern hardware," *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 4, pp. 709–730, 2002.

[6] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on modern processor architectures," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1754–1766, 2015.

[7] J. Kang, J. F. Naughton, and S. Viglas, "Evaluating window joins over unbounded streams," *ICDE*, pp. 341–352, 2003.

[8] J. W. Stamos and H. C. Young, "A symmetric fragment and replicate algorithm for distributed joins," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1345–1354, 1993.

[9] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch, "Scalable and adaptive online joins," *PVLDB*, vol. 7, no. 6, pp. 441–452, 2014.

[10] A. Okcan and M. Riedewald, "Processing theta-joins using mapreduce," *SIGMOD*, pp. 949–960, 2011.

[11] Q. Lin, B. C. Ooi, Z. Wang, and C. Yu, "Scalable distributed stream join processing," *SIGMOD*, pp. 811–825, 2015.

[12] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: fault-tolerant and scalable joining of continuous data streams," *SIGMOD*, pp. 577–588, 2013.

[13] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," *EDBT*, pp. 99–110, 2010.

[14] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," *SIGMOD*, pp. 975–986, 2010.

[15] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: reliable stream computation in the cloud," *EuroSys*, pp. 1–14, ACM, 2013.

[16] S. Wang and E. A. Rundensteiner, "Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing," *EDBT*, pp. 299–310, 2009.

[17] B. Gedik, R. Bordawekar, and P. S. Yu, "Celljoin: a parallel stream join operator for the cell processor," *VLDB J.*, vol. 18, no. 2, pp. 501–519, 2009.

[18] J. Teubner and R. Müller, "How soccer players would do stream joins," *SIGMOD*, pp. 625–636, 2011.

[19] P. Roy, J. Teubner, and R. Gemulla, "Low-latency handshake join," *PVLDB*, vol. 7, no. 9, pp. 709–720, 2014.

[20] D. L. Quoc, I. E. Akkus, P. Bhatotia, S. Blanas, R. Chen, C. Fetzer, and T. Strufe, "Approxjoin: Approximate distributed joins," *SoCC*, pp. 426–438, 2018.

[21] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: fault-tolerant streaming computation at scale," *SOSP*, pp. 423–438, 2013.

[22] C. Liu and H. Chen, "A hash partition strategy for distributed query processing," *EDBT*, pp. 373–387, 1996.

[23] A. Chakraborty and A. Singh, "Parallelizing windowed stream joins in a shared-nothing cluster," *CoRR*, vol. abs/1307.6574, 2013.

[24] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, "A holistic view of stream partitioning costs," *PVLDB*, vol. 10, no. 11, pp. 1286–1297, 2017.

[25] Y. Zhou, Y. Yan, F. Yu, and A. Zhou, "Pmjoin: Optimizing distributed multi-way stream joins by stream partitioning," *DASFAA*, pp. 325–341, 2006.

[26] F. Alvanaki and S. Michel, "Tracking set correlations at large scale," *SIGMOD*, pp. 1507–1518, 2014.

[27] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[28] P. Chardaire, M. Barake, and G. P. McKeown, "A probe-based heuristic for graph partitioning," *IEEE Trans. Computers*, vol. 56, no. 12, pp. 1707–1720, 2007.

[29] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," *ESA*, pp. 469–480, 2011.

[30] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs," *SC*, p. 28, 1995.

[31] "Apache Storm – distributed realtime computation system." http://storm.apache.org/.

[32] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," *SIGMOD*, pp. 147–156, 2014.

[33] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining (2Nd Edition)*. Pearson, 2nd ed., 2018.

[34] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *In SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, 2000.

[35] C. Chasseur, Y. Li, and J. M. Patel, "Enabling JSON document stores in relational systems," *WebDB 2013*, pp. 1–6, 2013.