Processing Interval Joins On Map-Reduce

Bhupesh Chawda, Himanshu Gupta, Sumit Negi, Tanveer A. Faruquie, L V Subramaniam, Mukesh Mohania
IBM India Research Laboratory, New Delhi, India
{bhchawda, higupta8, sumitneg, ftanveer, lvsubram, mkmukesh}@in.ibm.com

ABSTRACT

In this paper we investigate the problem of processing multi-way interval joins on map-reduce platform. We look at join queries formed by interval predicates as defined by Allen's interval algebra. These predicates can be classified in two groups: colocation based predicates and sequence based predicates. A colocation predicate requires two intervals to share at least one common point while a sequence predicate requires two intervals to be disjoint. An interval join query can therefore be thought of as belonging to one of the three classes: (a) queries containing only colocation based predicates, (b) queries containing only sequence based predicates and (c) queries containing both classes of predicates. We address these three classes of join queries, discuss the challenges and present novel approaches for processing these queries on map-reduce platform. We also discuss why the current approaches developed for handling join queries on real-valued data can not be directly used to handle interval joins. We finally extend the approaches developed to handle join queries containing multiple interval attributes as well as join queries containing both interval as well as non-interval attributes. Through experimental evaluations both on synthetic and real life datasets, we demonstrate that the proposed approaches comfortably outperform naive approaches.

1. INTRODUCTION

An interval is represented as the range $[t_s, t_e]$ which identifies the lifetime or the range of an event. An interval $[t_s, t_e]$ consists of a start point t_s and an end point t_e and includes all points in-between including t_s and t_e. Interval data is ubiquitous and a number of real-world scenarios generate huge volume of interval data. Consider spatio-temporal environment modeling data e.g., the measurements of weather attributes like pressure, temperature, rainfall, wind-speeds, pollutant-concentrations etc at different latitude, longitude, elevation and time-stamps. The time-stamps during which we observe high or low temperature, pressure, rainfall etc at a location are described using intervals. For example consider the event - 'A rainfall was observed during the period 7 AM and 7:15 AM'. Here the rainfall start and end time constitute the interval data. Consider telephonic call data - the timestamps during which a call happens is described using an interval. For example - 'A call was logged between 7 AM and 7:15 AM from cell number x.' Here the interval [7 AM, 7:15 AM] describes the duration of a call. Consider spatial data like buildings, rivers etc. The dimensions of the spatial objects are described using intervals. For example, consider a building at coordinates [100, 100] with length 20 and breadth 10 m. Here the interval [100, 120] describes the length of the building and the interval [100, 110] describes the breadth of the building.

Interval join is a key operation on interval data and is used to correlate intervals of different events. Consider again the spatio-temporal environment modeling data. From this data we construct intervals during which we observe high wind speed ($>$ a threshold s), high temperature ($>$ a threshold t) and high pollutant concentration ($>$ a threshold p). Consider the interval-join query - *For a given location, find all intervals u_1, u_2 and u_3 such that high wind speed, high temperature and high concentration of a pollutant were observed during intervals u_1, u_2 and u_3 respectively and the intervals u_2 and u_3 are contained within interval u_1'.* This involves joining intervals of high wind speed, high temperature and high pollutant concentration using the predicate 'contains'. Such results can be used for investigating various aspects towards building predictive models of the pollutant concentration. Alternatively consider spatial data describing cities, rivers etc and the query - 'Find all cities overlapping with a river'. This spatial join query involves joining the relations *cities* and *rivers* using the predicate *overlap* and further reduces to an interval join query - *select city from cities, river from rivers where city.length overlaps river.length and city.breadth overlaps river.breadth.*

In this paper we present novel approaches for handling interval joins on map-reduce platform [5]. Hadoop, an open-source implementation of map-reduce framework, has become the most popular platform for large-scale data analysis in a shared nothing parallel computing environment. A number of systems have been recently implemented on top of Hadoop for processing spatio-temporal data. These include *Spatial-Hadoop* [6], *Sci-Hadoop* [4], *CloST* [15] etc. The methods presented in this paper can be integrated with these systems for efficient processing of join queries on interval data. Related work carrying out a spatio-temporal data analysis on Map-Reduce includes SystemML [7], k-NN

10.5441/002/edbt.2014.42

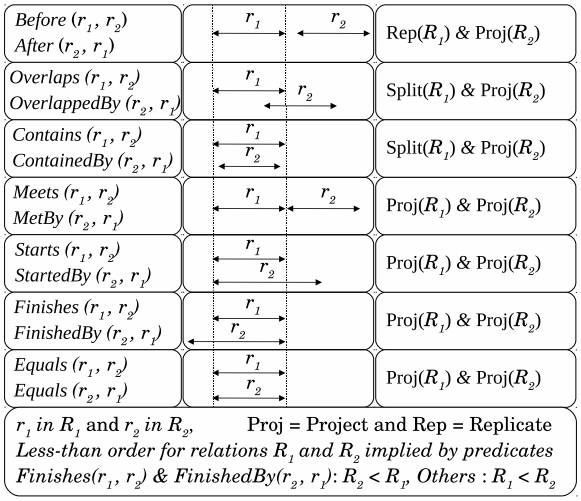


Figure 1: Allen's Interval Algebra

joins [11], Trajectory data analysis [12], Spatial-Joins [8] etc.

Computing a multi-way join on map-reduce is not a simple task. Consider a multi-way join query $\mathcal{Q}$ formed by m relations $\{R_1, R_2, \ldots, R_m\}$. Consider an output tuple $(u_1, u_2, \ldots, u_m)$ where tuple u_i belongs to relation R_i. Since the data for each relations resides in a different file on HDFS, the m input tuples are processed by m different map processes. To compute an output tuple, it is required to bring together all m tuples from m different map processes to at least one reduce node; and this must happen for each output tuple. A tuple u_i in relation R_i will hence be communicated to multiple reduce nodes. Larger the traffic between map and reduce nodes, larger is the cost of the map-reduce algorithm. If there are M reduce nodes, the goal is to devise an algorithm which generates minimal intermediate traffic between map and reduce processes while at the same time ensuring that all M reduce nodes receive similar volume of load to process. A number of studies have investigated this problem for different classes of join queries. These include 2-way theta join [14], multi-way equality joins [2], K-NN joins [11], spatial joins [8], multi-way theta joins [17], self-similarity joins [16, 13] etc. In this paper we look at this problem for the case of multi-way interval join queries.

Contribution: Specifically we investigate interval join queries involving predicates in Allen's interval algebra [3]. Allen's algebra [3] is a calculus for reasoning with intervals which defines possible relations between intervals and provides a basis for reasoning about temporal descriptions of events. Figure 1 outlines the relations between two intervals in Allen's algebra. The two intervals are represented using variables r_1 and r_2. For all predicates other than *before* and *after*, one of the start, end or body of an interval co-occurs with the start, end or the body of another interval. Such predicates are termed colocation based predicates. The predicates *before* and *after* are not based on colocation of intervals but rather on sequencing of intervals; and hence termed sequence based predicates. Colocation based predicates can be *likened* to equality predicates while sequence based predicates can be *likened* to theta/inequality predicates. The contributions of this paper can hence be summarized as follows:

- We classify the interval join queries in four classes - (1) Colocation i.e., queries involving a single interval attribute and only colocation predicates, (2) Sequence i.e., join queries involving a single interval attribute and only sequence predicates, (3) Hybrid i.e, queries involving a single interval attribute and both colocation and sequence predicates and (4) General i.e, queries involving one or more than one interval attributes; queries may also involve real-valued attributes; the predicates may be colocation or sequence in case of interval attributes while equi or theta predicates in case of real-valued attributes.

  Colocation, Sequence and Hybrid queries are hence a special case of general queries. We present novel algorithms for handling each of these interval join query classes on map-reduce. To the best of our knowledge, this is the first study to discuss join processing on interval datasets on map-reduce platform.

- An interval is a collection of points with a marked start-point as well as an end-point. An interval is hence fundamentally different from a real-valued data point. A real-valued data point is an interval of length 0. Similarly note that as the intervals are reduced to length 0, all colocation predicates in Allen's algebra reduce to equality predicates on real-valued data while all sequence predicates reduce to inequality predicates on real-valued data. Handling interval data is hence more complex vis-a-vis real-valued data. Through-out this paper we delve into this aspect in detail and clearly outline the nature of this complexity. We outline both naive as well as optimized approaches of handling this additional complexity vis-a-vis real-valued data.

- To handle a multi-way join, the data/reducers are visualized as part of a multi-dimensional space. This is done to achieve a better load-balance among the reducers as opposed to when data/reducers are considered as part of a single-dimensional space. Map operations process intervals and communicate each interval to one or more reducers. Each reducer computes a part of the output from the input intervals it receives. For each of the four classes of interval join queries, we hence answer the following questions clearly:
  - In how many dimensions do we visualize data / reducers?
  - Which of the reducers in this multi-dimensional space won't produce any output? We can then apriori avoid communicating any data to such reducers. We call such reducers, inconsistent reducers. We argue that depending on which Allen's predicates are present in the multi-way join query, we can apriori identify a subset of the inconsistent reducers.
  - For each input-tuple in each relation we identify the set of consistent reducers, to which this tuple should be communicated to.
  - Which reducer in the multi-dimensional space will compute a given output tuple?

  We introduce the notion of *less-than-order* as implied by Allen's predicates which helps in answering these questions. We discuss this notion in detail and discuss how we exploit this to design efficient approaches for handling multi-way interval join queries.

- We carry out a detailed experimental evaluation over both synthetic and real-life data to validate the ideas presented in this paper. We show that the algorithms presented in this paper comfortably out-perform the naive approaches.
- We clearly position our work vis-a-vis related works in the domain of multi-way join on real-valued data. There are three main related works - Work [14] handling 2-way theta-join queries on real-valued data, Work [2] handling multi-way equi-join queries on real-valued data and Work [17] handling multi-way theta joins queries on real-valued data. We discuss the differences and the complementary nature of these studies vis-a-vis ours, wherever appropriate.

Organization: Section 2 presents the map-reduce model and the general strategy for processing a join query. Section 3 develops the notation of project, split and replicate operations on interval data. These operations form the building block operations of the algorithms developed in this paper. Section 4 discusses how we handle 2-way interval joins. Section 5 develops three key concepts - (a) *less-than-order* among relations, (b) consistent interval-sets and (c) crossing interval-sets. Section 6, 7, 8 and 9 present novel approaches for handling multi-way colocation, sequence, hybrid and multi-attribute interval joins respectively. Section 10 mentions the related work. Section 11 concludes the paper.

2. HADOOP, MAP-REDUCE AND JOIN

Hadoop Distributed File System (HDFS) manages the storage of data on Hadoop. Data is read from HDFS and is passed to map functions for processing. The default method is to read data line-by-line from HDFS files and pass each line to the map functions for processing. A map-function processes each line and converts this line into a set of intermediate key-value pairs. Hadoop then collects these pairs and communicates these pairs to reduce functions in a fashion so as all pairs with the same key are communicated to the identical reducer. Multiple map and reduce processes can parallely run on a set of machines. For more details of the map-reduce framework we refer the reader to [5].

To develop a map-reduce (MR) algorithm for a task, one needs to determine - what forms the key and what forms the value (as part of intermediate key-value pairs). Larger the number of intermediate pairs, larger is the communication cost among map and reduce functions. In case of a multi-way join, all relations are stored as separate HDFS files and each line usually represents a tuple. Hadoop reads-in each tuple u from these relations and passes the tuple u to the map-function. A map-function converts this tuple to a set of intermediate key-value pairs $\{\langle$*reducer-id*,$u\rangle\}$. A pair $\langle$*reducer-id*,$u\rangle$ implies that the tuple u is communicated to a reduce process with id *reducer-id*. A map-function hence communicates a tuple to one or more reducers. A reduce process with id r hence receives all tuples which have been sent by map functions as part of pairs with key equal to r. Reducer r computes the join among the tuples it receives. This join output is hence a partial join output. Combining the output of all reducers, generates the whole join output.

A join MR algorithm hence must satisfy the following: *Consider there are m relations in the join query and hence each output tuple consists of m input-tuples, one from each relation. For each output tuple, all m input tuples must be received by at least one reducer.* If this is not so for an output tuple then this output tuple can not be computed by any reducer and hence the join implementation is incorrect.

A good join MR algorithm optimizes the following criteria - (a) The number of intermediate key-value pairs should be minimal and (b) the volume of pairs received by all reducers should be similar. If condition (b) is not satisfied, then reducers receiving larger volume of traffic will run for a long period of time while reducers receiving smaller volume of traffic will finish quickly, thereby leading to an inefficient use of cluster resources and hence an inefficient join MR implementation. Criteria (a) and (b) are optimized by exploiting the properties of the input-data (e.g., the data consists of intervals or sets or real-valued points etc), statistical distribution of the input data (uniformly distributed data vs skewed data will need to be processed differently) , properties of the join predicates (equi join or theta join) etc.

3. PROJECT-SPLIT-REPLICATE

Partitioning: Let the complete time range be $[t_0,t_n)$ i.e., all intervals in the relation lie within this range. A partitioning of time range $[t_0,t_n)$ is defined as a sequence of contiguous intervals $([t_{i0},t_{i1}),[t_{i1},t_{i2}),\ldots,[t_{i(l-1)},t_{il}))$, $t_{i0} = t_0$ and $t_{il} = t_n$. Each of these intervals is called a partition-interval. We equivalently represent the partitioning as $\mathcal{P}=(p_1,p_2,\ldots,p_l)$ where the partition-interval p_j represents the interval $[t_{i(j-1)},t_{ij})$.

Project: Project operation determines the partitioning interval in which the start point of an interval lies.The projection of an interval u having range $[t_s,t_e]$ on a partitioning $\mathcal{P}$ results in the generation of a single key-value pair (p_i, u) where p_i is the partition in which the start-point of the interval lies.

$$\text{Project}(u,\mathcal{P}) \rightarrow \{(p_i,u) \mid u.t_s \in p_i\}$$

Split: Split operation returns all the partitions which have at-least one point in common with the interval. For each such partition, a key-value pair is generated and hence a set of key-value pairs is returned for each interval.

$$\text{Split}(u,\mathcal{P}) \rightarrow \{(p_i,u) \mid u \cap p_i \neq \phi \}$$

Replicate: The replicate operation returns all the partitions which have at least one point which is greater than or equal to the start-point of the interval. Formally,

$$\text{Replicate}(u,\mathcal{P}) \rightarrow \{(p_i,u) \mid u \cap p_i \neq \phi \vee u.t_s < p_i.t_s \}$$

Projecting, Splitting and Replicating a relation: Equivalently we define projecting, splitting and replicating a relation R as projecting, splitting and replicating all intervals in relation R respectively. Consider Figure 2. The relation R here consists of two intervals u and v. The partitioning $\mathcal{P}$ has four partition-intervals. The result of projecting, splitting and replicating relation R is mentioned in Figure 2. Intervals u and v start in partition-intervals p_1 and p_2 respectively. The project output hence consists of two pairs (p_1,u) and (p_1,v). Interval u overlaps with p_1 and p_2 and hence splitting u outputs two pairs - (p_1,u) and (p_2,u). Interval v overlaps only with p_2 and hence splitting v outputs only one pair (p_2,v). All four partition-intervals p_1,p_2,p_3 and p_4 have one point greater than or equal to the start-point of the interval u and hence replicating u outputs four pairs. Replicating v outputs three pairs as only p_2,p_3 and p_4 have one point greater than or equal to the start-point of the interval v.

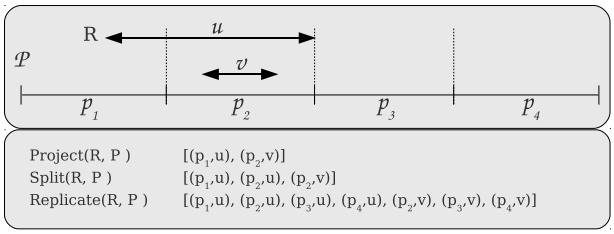


Figure 2: Project, Split and Replicate Example

Intuition: A map function processes an interval (tuple) by either projecting, splitting or replicating the interval. Partition-intervals p_i's also denote the reducer-ids. A map function generating an intermediate key-value pair (p_i, u) implies that the map-function is communicating interval u to reducer p_i.

Loosely speaking, we can visualize the time-range $[t_0, t_n)$ to be the whole computation and each partition-interval p_i as a partial computation. The computation for partition-interval p_i is carried out by reducer p_i. Reducer p_i processes all intervals intersecting with partition-interval p_i plus some more intervals intersecting with other partition-intervals $p_j, j \neq i$.

4. 2-WAY INTERVAL JOINS

The basic idea in writing a map-reduce program for a join is to get the tuples agreeing on the join predicate on the same reducer. In this section we describe how we achieve this in case of 2-way interval joins. Let the number of reducers be k. We first divide the complete time-range into k partition-intervals. Depending upon the exact Allen's predicate being used, we either project, split or replicate the relations.

Colocation predicate - *Overlap* Consider two relations R_1 and R_2. Let $r_1 \in R_1$ and $r_2 \in R_2$ be two tuples which satisfy the *Overlaps* predicate (Figure 1). An interval r_1 overlaps interval r_2 if the start-point of r_2 lies between the start-point of r_1 and the end-point of r_1 (and not the other-way round. *overlap* and *overlapped-by* are two different Allen's predicates; Figure 1).

Overlaps(R_1,R_2) can be computed by splitting R_1 and projecting R_2. Let's suppose the tuple r_2 is projected on the partition interval p_i. The project operation on interval r_2 will hence produce the pair (p_i, r_2). As r_1 overlaps r_2, the output of splitting r_1 will consist of the pair (p_i, r_1). Hence these two tuples will be present at the reducer p_i. The reducer p_i can hence output the joined tuple (r_1, r_2). This output tuple will not be produced by any other reducer as the tuple r_2 is routed only to reducer p_i. It is because the relation r_2 is being projected and the project operation produces only one key-value pair for every interval.

Sequence predicate - *Before* *Before*(R_1,R_2) is computed by projecting R_2 and replicating R_1. As *Before* is a sequence predicate, an interval will match all the intervals occurring after it. To ensure that every such pair is present at at-least one reducer, relation R_1 needs to be replicated.

Similarly we can argue about all Allen's predicates. Figure 1 outlines all the Allen's predicates and the corresponding operations required to carry out the interval join (Column 3).

5. LESS-THAN-ORDER, CONSISTENT INTERVAL SETS AND CROSSING INTERVAL-SETS

This section develops the notions of *less-than order*, *consistent* interval-sets and *crossing* interval-sets. Approaches presented in this paper are based on these concepts.

5.1 Less-Than-Order

Less-Than Order between two Intervals: An interval u=(t_{s1},t_{e1}) is said to be in less-than order with interval v=(t_{s2},t_{e2}) if t_{s1} is less than or equal to t_{s2}. Equivalently we say that interval u is less than interval v or interval u is in less-than order relationship with interval v.

Given a set of intervals $\mathcal{U}$=$\{u_1$,u_2,...,$u_n\}$ the interval u_i is said to be the left-most or right-most interval if the start-point of this interval is the least or maximum of all intervals. There can be more than one left-most or right-most intervals if they start at the same point.

Consider the Figure 3. The interval u_1 is less-than v_1 as u_1 starts before v_1 does. u_1 is the left-most interval as the start-point of u_1 is less than the start-point of all other intervals. u_5 is the rightmost interval as the start-point of u_5 is larger than the start-point of all other intervals.

Less-Than Order between two Relations: Consider the 2-way join query $R_1 P R_2$. We say that the predicate P enforces a less-than-order between R_1 and R_2 if for each interval pair (u, v), $u \in R_1$, $v \in R_2$ which satisfies predicate P, the interval u is in less-than-order relationship with v. Similarly we say that the predicate P enforces a less-than-order between R_2 and R_1 if for each interval pair (u, v), $u \in R_1$, $v \in R_2$ which satisfies predicate P, the interval v is in less-than-order relationship with u. Note that the 2-way join query remains the same i.e., $R_1 P R_2$.

For example, consider $R_1 Overlaps R_2$. The predicate *Overlap* enforces a less-than order between R_1 and R_2 as the interval from relation R_1 must start before interval from relation R_2. All of Allen's predicates enforce a less-than order between its two relations. Figure 1 mentions the less-than-order relationships for all the Allen's predicates.

5.2 Consistent Interval-Sets

Consider a query $\mathcal{Q}$ and its relation-set $\mathcal{R}$. A set of intervals $\mathcal{U}$ is called a consistent interval-set if for each pair of intervals u and v in $\mathcal{U}$ the following conditions hold:

- **A1:** Intervals u and v belong to different relations.
- **A2:** Consider u and v belong to R_u and R_v, $R_u \in \mathcal{R}$, $R_v \in \mathcal{R}$. If there is a condition $R_u P R_v$ in query $\mathcal{Q}$ then intervals u and v satisfy the predicate P. Similarly if there is a condition $R_v P R_u$ in query $\mathcal{Q}$ then intervals v and u satisfy the predicate P.

Note that each subset of a consistent interval-set is also consistent. Consider the query $\mathcal{Q}_0$: R_1 *overlaps* R_2 *and* R_2 *contains* R_3 *and* R_3 *overlaps* R_4 in Figure 3. $\mathcal{Q}_0$'s relation-set is $\mathcal{R}$=$\{R_1$,R_2,R_3,$R_4\}$. Intervals from R_1, R_2, R_3 and R_4 are denoted by u, v, w and x respectively. The interval-set $\mathcal{U}_1$=$\{u_3$,v_1,$w_1\}$ is consistent. $\mathcal{U}_1$'s relation-set is $\mathcal{R}_1$=$\{R_1$, R_2, $R_3\}$. There are two conditions in query $\mathcal{Q}_0$ which are formed by the relations in $\mathcal{R}_1$ i.e., R_1 *overlaps* R_2 and R_2 *contains* R_3. The condition R_1 *overlaps* R_2 requires that the intervals u_3 and v_1 overlap and they do. The condition R_2 *contains* R_3 requires that the interval v_1 contains the interval w_1 and it does. Hence the interval-set $\mathcal{U}_1$ satisfies both conditions A1 and A2 for consistency. Similarly $\mathcal{U}_2$=$\{u_2$,v_1,w_1,$x_3\}$ is a consistent interval-set. The interval-set $\mathcal{U}_3$=$\{u_1$,v_1,$w_1\}$ is not consistent as the interval u_1 does

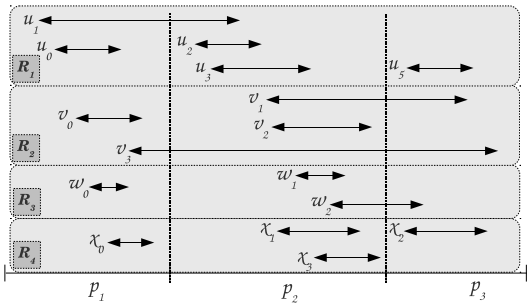


Figure 3:

not overlap with v_1 while that is required due to the presence of condition R_1 *overlaps* R_2 in $\mathcal{Q}_0$.

5.3 Crossing Interval-Sets

Consider a query $\mathcal{Q}$ and its relation-set $\mathcal{R}$. Consider an interval-set $\mathcal{U}$ and its relation-set $\mathcal{R}_\mathbf{U}$. The set $\mathcal{U}$ crosses the partition-interval p if the following conditions hold:

- No two intervals in $\mathcal{U}$ belong to the same relation.
- Each interval u in $\mathcal{U}$ intersects with partition-interval p i.e. interval u and partition-interval p have at least one point in common.
- For all join conditions R_1 P R_2 or R_2 P R_1 in query $\mathcal{Q}$ s.t. $R_1 \in \mathcal{R}_\mathbf{U}$ and $R_2 \in \mathcal{R} - \mathcal{R}_\mathbf{U}$, the following holds. Let u be the interval in $\mathcal{U}$ belonging to relation R_1.
  - **B1:** If the predicate P enforces a less-than order between R_1 and R_2 then the interval u crosses the right-boundary of partition-interval p i.e., the end-point of the interval u lies in any partition-interval following p_i.
  - **B2:** If the predicate P enforces a less-than order between R_2 and R_1 then the interval u crosses the left-boundary of partition-interval p i.e., the start-point of the interval u lies in any partition-interval preceding p_i.

Consider the interval-set $\mathcal{U}_4$={u_3,v_1,w_2} in Figure 3 and the query $\mathcal{Q}_0$. $\mathcal{U}_4$ crosses partition-interval p_2 as it satisfies all the four conditions for crossing p_2. All three intervals in $\mathcal{U}_4$ belong to different relations and intersect with partition-interval p_2. The relation-set of $\mathcal{U}_4$ is $\mathcal{R}_{\mathbf{U}_4}$={$R_1$, R_2, R_3}. The relation-set of query $\mathcal{Q}_0$ is {R_1,R_2,R_3,R_4}. The set $\mathcal{R}$-$\mathcal{R}_{\mathbf{U}_4}$ hence is {R_4}. There is only one condition in query $\mathcal{Q}_0$ with one relation in set $\mathcal{R}_{\mathbf{U}_4}$ and the other relation in set $\mathcal{R}$-$\mathcal{R}_{\mathbf{U}_4}$ i.e., R_3 overlaps R_4. As *overlap* predicate enforces R_3 to be less than R_4, the condition B1 requires that the interval belonging to relation R_3 in $\mathcal{U}_4$ (i.e., w_2) crosses the right boundary of partition-interval p_2 and w_2 does. No other interval is required to cross either the left or right boundary of p_2.

The interval-set $\mathcal{U}_5$={v_3,w_2} also crosses the partition-interval p_2. Its relation-set is $\mathcal{R}_{\mathbf{U}_5}$={$R_2$, R_3}. The set $\mathcal{R}$ - $\mathcal{R}_{\mathbf{U}_5}$ is {R_1, R_4}. There are two conditions in query $\mathcal{Q}_0$ with one relation in set $\mathcal{U}_5$ and the second in $\mathcal{R}$ - $\mathcal{R}_{\mathbf{U}_5}$. The condition R_1 *overlaps* R_2 requires the interval v_3 to cross the left-boundary of p_2 and it does. The condition R_3 *overlaps* R_4 requires w_2 to cross the right boundary of p_2 and it does. The interval-set $\mathcal{U}_6$={v_3,w_1} does not cross the partition-interval p_2 as the interval w_1 does not cross the right boundary of p_2 while that is required due to the condition R_3 *overlaps* R_4 .

Intuition: An output tuple is always a consistent interval-set and all its subsets are also consistent. If intervals in an output tuple $\mathcal{U}$ span more than one partition-intervals, $\mathcal{U}$ can be visualized as a union of consistent interval sets some of which also cross some partition-intervals. A consistent interval-set which crosses a partition-interval can combine with another consistent interval-set to form an output tuple. For example consider the output tuple {u_2, v_1, w_2, x_2} in Figure 3. It can be seen as a union of two interval-sets - $\mathcal{U}_1$={u_2, v_1, w_2} and $\mathcal{U}_2$={x_2}. $\mathcal{U}_1$ is consistent, crosses the partition-interval p_2 and combines with $\mathcal{U}_2$ to form the output tuple. In this paper we propose efficient algorithms for computing multi-way interval colocation joins which work by tracking consistent and crossing interval-sets.

6. COLOCATION MULTI-WAY JOINS

In this section we present *Replicate Consistent And Crossing Interval Sets* (RCCIS), an efficient algorithm for handling multi-way interval colocation joins involving single interval attribute. We first present two naive approaches.

2-way Cascade (*2-way Cd*) This processes a multi-way join query as a series of 2-way joins. For example again consider the query $\mathcal{Q}_0$ - R_1 *overlaps* R_2 *and* R_2 *contains* R_3 *and* R_3 *overlaps* R_4. This approach will first join R_1 and R_2 for predicate *overlap*. Then it will join the result of this 2-way join with relation R_3 for predicate *contains*. Finally the result will be joined with R_4 for predicate *overlap*.

All-Replicate (*All-Rep*): A colocation join query can be handled by projecting the rightmost relation and replicating the rest. For example for $\mathcal{Q}_0$, we can replicate R_1, R_2, R_3 and project R_4. Any output tuple {u,v,w,x} is hence computed at the reducer on which the interval x is projected. For queries which contain more than one rightmost relations, all relations need to be replicated.

Note that splitting all relations does not work. For example consider the query $\mathcal{Q}_0$, Figure 3 and the output tuple $\mathcal{V}$={u_3, v_1, w_2, x_2}. As a result of splitting all these four intervals in $\mathcal{V}$, the reducer p_1 receives no interval, reducer p_2 receives intervals {u_3, v_1, w_2} and the reducer p_3 receives intervals {v_1, w_2, x_2}. Hence none of the three reducers receive all the four intervals and hence this output tuple can not be computed. Replicating all relations but one is a naive way of ensuring that for each output tuple, all its constituents intervals will be received by a reducer.

Both these approaches are inefficient. *2-way Cd* produces a series of big intermediate join results and these results are joined with subsequent relations. Each join is executed as a new set of map and reduce tasks and these tasks involve reading a lot of data and communicating it to the reducers. *2-way Cd* hence involves huge reading and communication cost. *All-Rep* replicates all intervals in all but one relations and this also implies a huge communication cost. We next present the algorithm RCCIS which improves on both these approaches substantially.

6.1 Replicate Consistent And Crossing Interval Sets (RCCIS)

Basic Goal: We need an approach which solves a multi-way join in one go rather than as a cascade of 2-way joins but unlike *All-Replicate* which does not replicate all intervals. This is what precisely the *RCCIS* approach achieves. It

selectively identifies which intervals need to be replicated and replicates only such intervals.

Outline: *RCCIS* runs in two cycles of map-reduce. The first round of map operations split all the relations. The reducer p_i hence receives all intervals that intersect with partition-interval p_i. Reducer p_i then runs the *RCCIS* algorithm (described in detail next) on the intervals received and decides which intervals need to be replicated. Reducer p_i then writes out all the intervals on the disk along-with a flag to indicate if the interval was selected for replication. The second round of map operations read the output of first round of reducers and replicate the intervals selected for replication and project the rest. Second round of reducers then carry out the join among the intervals obtained.

Details: We now describe the strategy reducer p_i follows in the first round to select the intervals to replicate. Let the query be $\mathcal{Q}$ and its relation-set $\mathcal{R}$. Let $\mathcal{U}_{p_i}$ be the set of intervals split on partition interval p_i in the 1^{st} round and subsequently received by reducer p_i. The reducer p_i first identifies all interval-sets $\mathcal{U}$, $\mathcal{U} \subset \mathcal{U}_{p_i}$ s.t. interval-set $\mathcal{U}$ satisfies the following conditions.

1. **C1:** $\mathcal{U}$ is a consistent interval-set.
2. **C2:** $\mathcal{U}$ crosses the partition-interval p_i.

Let $\mathcal{US}_{p_i}$ be the set of such interval-sets. Let uS_{p_i} represent the set of all intervals which belong to any interval-set in $\mathcal{US}_{p_i}$ i.e., uS_{p_i} is the union of all interval-sets in $\mathcal{US}_{p_i}$.

RCCIS replicates all intervals in uS_{p_i} which start in partition interval p_i i.e., the start-point t_s of interval u lies in partition-interval p_i. We omit a proof-of-correctness due to space constraints.

Intuition behind RCCIS: $\mathcal{U}_{p_i}$ is the set of intervals received by reducer p_i in the first round. The objective of *RCCIS* is to identify which intervals in $\mathcal{U}_{p_i}$ need to be replicated and which need not be. *All-Rep* replicates all intervals and hence is an inefficient algorithm.

To compute an output tuple $\mathcal{V}$, all intervals in $\mathcal{V}$ must be brought to a reducer. Consider an output tuple $\mathcal{V}$=$\{u_1, u_2, \ldots, u_m\}$, $u_i \in R_i$. If a reducer p_i receives all intervals in $\mathcal{V}$ in the first round, the reducer p_i can compute this output tuple. No interval in $\mathcal{V}$ hence needs to be replicated. For example, consider the interval-set $\{u_0,v_0,w_0,x_0\}$ and query $\mathcal{Q}_0$ in Figure 3. Reducer p_1 receives all the four intervals and hence can compute this output tuple. *All-Replicate* replicates all intervals and is hence clearly redundant. Note that an output tuple is not a crossing-set (Section 5.3) and hence does not satisfy the condition C2 of *RCCIS*.

However this may not be the case and reducer p_i may receive only some of the intervals in an output tuple $\mathcal{V}$. Reducer p_i hence needs to replicate the intervals so that one following reducer (i.e. p_j, $j > i$) can receive all these intervals in the second round and can hence compute the output tuple. However this requires first identifying which intervals in $\mathcal{U}_{p_i}$ may form a part of an output tuple. The conditions C1 and C2 of *RCCIS* identify such intervals.

RCCIS constructs interval-sets out of set $\mathcal{U}_{p_i}$ which can be part of an output tuple. A set which is not consistent can not be part of an output tuple. Further a consistent set which does not cross partition-interval p_i can not combine with any other set from other partition-intervals to form an output tuple. Reducer p_i hence computes sets which are consistent as well as cross the partition-interval p_i. The set $\mathcal{US}_{p_i}$ consists of such interval-sets.

uS_{p_i} consists of intervals belonging to any set in $\mathcal{US}_{p_i}$. Reducer p_i replicates those intervals in uS_{p_i} which start in partition-interval p_i. Intervals not starting in partition-interval p_i will possibly be replicated by reducers preceding p_i. As the number of intervals replicated by *RCCIS* is much smaller vis-a-vis *All-Rep* which replicates all intervals, *RCCIS* improves considerably over *All-Rep*.

Computing Output Tuple: Consider an output tuple $\mathcal{U}$=$\{u_1,u_2,\ldots,u_m\}$. The output tuple $\mathcal{U}$ is computed at the reducer (say p_j), on which the right-most interval in $\mathcal{U}$ is projected on. If some intervals in $\mathcal{U}$ do not intersect with partition-interval p_j, they will be received by reducer p_j as a result of selected-replication by reducers preceding p_j (in the first round).

Example: Let's consider Figure 3 and the query $\mathcal{Q}_0$ again. For the intervals presented in Figure 3, the output will consist of six tuples $\mathcal{V}_1$=$\{u_3,v_1,w_2,x_2\}$, $\mathcal{V}_2$=$\{u_3,v_1,w_1,x_3\}$,$\mathcal{V}_3$=$\{u_3,v_2,w_1,x_3\}$, $\mathcal{V}_4$=$\{u_1,v_3,w_2,x_2\}$,$\mathcal{V}_5$=$\{u_1,v_3,w_1,x_3\}$ and $\mathcal{V}_6$=$\{u_0,v_0,w_0,x_0\}$. We now illustrate *RCCIS* approach for reducer p_2. Reducer p_2 receives the intervals $\mathcal{U}_{p_2}$=$\{u_1, u_2, u_3, v_1, v_2, v_3, w_1, w_2, x_1, x_3\}$ in the first round as a result of splitting all relations. The set $\mathcal{US}_{p_2}$ is $\{ \{u_3,v_1,w_2\}, \{v_3,w_2\} \}$ as only these two interval-sets satisfy conditions C1 and C2. The set uS_{p_2} is hence $\{u_3,v_1,v_3,w_2\}$ and reducer p_2 replicates intervals in uS_{p_2} which start out in partition-interval p_2 i.e., $\{u_3,v_1,w_2\}$.

Discussion: As discussed in section 5, the interval-set $\mathcal{U}_1$=$\{u_3,v_1,w_2\}$ is consistent and crosses partition-interval p_2. $\mathcal{U}_1$ is hence included in the set $\mathcal{US}_{p_2}$. This signifies that the set $\mathcal{U}_1$ could combine with a consistent set in partition-interval p_3 (or any p_j , j > 2) to form an output tuple of the form $\{u_3,v_1,w_2, x\}$, $x \in R_4$, x starts after p_2 finishes. $\mathcal{V}_1$=$\{u_3,v_1,w_2,x_2\}$ is indeed such an output tuple and is computed by reducer p_3. Reducer p_3 receives intervals in $\mathcal{U}_1$ in the second round as a result of replication by reducer p_2 and receives interval x_2 as a result of projection by second round of map operations. Supposing the interval x_2 was not present, the reducer p_2 would still have needed to replicate intervals in $\mathcal{U}_1$ as the reducer p_2 has got no way to know this apriori.

As discussed in section 5, the interval-set $\mathcal{U}_2$=$\{v_3,w_2\}$ is also consistent and crosses the partition-interval p_2. Interval-set $\mathcal{U}_2$ is hence included in $\mathcal{US}_{p_2}$. Reducer p_2 needs to include $\mathcal{U}_2$ in $\mathcal{US}_{p_2}$ as there might be an output tuple of the form $\{u,v_3,w_2,x\}$ s.t. $u \in R_1$, $x \in R_4$, u finishes before p_2 starts and x starts after p_2 finishes. $\mathcal{V}_4$=$\{u_1,v_3,w_2,x_2\}$ is such an output tuple. Note that the interval u_1 is replicated by reducer p_1, intervals v_3 and w_2 replicated by reducer p_2 and the output tuple $\mathcal{V}_4$ is computed by reducer p_3.

The interval-set $\mathcal{U}_3$=$\{u_3,v_1,w_1\}$ is consistent but does not crosses the partition-interval p_2 as interval w_1 does not cross the right boundary. Hence there can not be an output tuple $\{u_3,v_1,w_1,x\}$ s.t. $x \in R_4$ and the start-point of x lies after the partition-interval p_i finishes. The interval-set $\mathcal{U}_3$ is hence not included in $\mathcal{US}_{p_2}$. This set violates the condition C2.

6.2 Experimental Evaluation

Hadoop Cluster SetUp: Experiments are run over a 16 core Hadoop cluster (v0.20.2) built using Blade Servers with four 3 GHz Xeon processors having 8GB memory and 200 GB SATA drives. The machines run Red Hat Linux 5.2.

Generation of Synthetic Data: We write a script to generate a set of intervals. The parameters to this script

are: (a) Number of intervals (nI), (b) Distribution of start points of intervals (dS). (c) Distribution of interval length (dI), (d) Range of time-points within which all intervals lie (t_{min}, t_{max}), (e) Min and max interval lengths (i_{min}, i_{max}).

Details of Real-life Data: We use publicly available Internet packet traces, collected from WIDE, a 150 Mbps trans-pacific Internet backbone [1]. For each day, a 15 minute extract is made public for download. This backbone carries Internet traffic between US and Japan. A packet trace consists of the values of all fields in IP and TCP headers of all packets passing through an observation point.

Given such a trace, we construct source-destination packet trains. A packet train consists of the sequence of packets flowing from a source IP to a destination IP such that the difference between two packet arrivals (at the observation point) is less than a threshold. Such trains are used in building network traffic models [9]. We hence have for each packet train its start-time i.e., the arrival time of the first packet of the train at the observation point and its end-time i.e., the arrival time of the last packet of the train. The durations of the packet trains hence form our interval data.

Experiments on Synthetic Data: We consider the query $\mathcal{Q}_1$=R_1 *overlaps* R_2 *and* R_2 *overlaps* R_3. All experiments are run with 16 reduce processes. Relations R_1, R_2 and R_3 are generated synthetically. Table 1 presents a comparison of all three approaches. Parameter values used for generating synthetic data are also provided. All three relations are of same size and the size is increased in steps of 0.25M. *RCCIS* easily beats both *2-way Cd* and *All-Rep*. Table 1 also presents the number of intervals replicated by *RCCIS* and *All-Rep*. Note that *RCCIS* replicates much smaller number of intervals vis-a-vis *All-Replicate* to compute the join output. As *All-Rep* replicates all intervals, it incurs a huge communication cost. The numbers in brackets represent the total number of key-value pairs generated after replication (if interval u is replicated to n reducers, u is counted n times). Note that this number is high in case of *All-Rep* because it replicates all intervals and high in case of *2-way Cd* as it reads big intermediate results. Comparatively this number is small for *RCCIS*. As a result, *RCCIS* incurs much smaller communication cost as well as much smaller cost for computing partial join output by each reducer in 2^{nd} round. We also carried out experiments varying other parameters like distribution of start-point of intervals (dS), max interval length (i_{max}) etc and we observed similar results. We do not outline the details due to the lack of space.

Table 1: Varying Data Size

dS,dI=Uniform, (t_{min},t_{max})=(0,100K), (i_{min}, i_{max})=(1,100)						
nI (M)	Time *2-way Cd.* (hh:mm)	Time *All-Rep* (hh:mm)	Time *RCCIS* (hh:mm)	# Intervals Replicated *RCCIS*	# Intervals Replicated *All-Rep*	# Pairs 2-way Cd
0.5	00:18	00:06	00:03	14.7K,(3.1M)	1M,(10.5M)	(84.6M)
0.75	00:43	00:28	00:06	21.8K,(4.7M)	1.5M,(15.8M)	(188.5M)
1.0	01:19	00:48	00:09	29.2K,(6.2M)	2.0M,(21.1M)	(334.4M)
1.25	02:07	01:05	00:15	36.6K,(7.8M)	2.5M,(26.4M)	(517.2M)

Experiments on Internet Packet Trace Data: We next showcase the efficacy of *RCCIS* on Internet packet trace data. We choose six 15 minute long traces collected across links SamplePointB and SamplePointF from MAWI repository, one each from year 2003 to 2008 in the month of January. The six traces are so chosen that they contain widely different number of packets and hence different statistical characteristics. Table 2 lists out the chosen traces

Table 2: Results on Internet Packet Trace Data

Trace	Date (dd-mm-yy)	# Pkts	# Pkt Trains	# Copies & Total Duration (min)	Time *2-way Cd.* (hh:mm)	Time *RCCIS* (hh:mm)
P03	01-01-03	1.5M	120K	25, 375	00:24	00:07
P04	01-01-04	0.2M	18K	167, 2500	00:13	00:06
P05	15-01-05	2.9M	207K	15, 225	00:35	00:07
P06	01-01-06	3.4M	351K	9, 135	01:03	00:08
P07	15-01-07	9.1M	359K	9, 135	01:22	00:09
P08	01-01-08	7.3M	307K	10, 150	02:08	00:11

and the number of packets in direction from Japan to US. Number of packets in the six traces vary from 0.2 million to 9.1 million. From these traces, we construct packet trains with inter-arrival cut-off being 500 ms. Table 2 also presents the number of packet trains generated for each trace. For each set of packet-trains computed, we generate a larger data containing 3 million packet trains by replicating the original data. We then compute the star self-join with *overlaps* predicate (i.e. R *overlaps* R and R *overlaps* R) on this 3M packet trains data with the number of reducers being 16. This finds out all triples {T1, T2, T3} such that train T1 overlaps with T2 and T2 overlaps with T3. We again find that *RCCIS* easily outperforms *2-way Cd.* on all the six packet traces.

6.3 Discussion

Lets consider the scenario where all the intervals are of length 0 i.e., interval data reduces to real-valued data. In such a case, multi-way colocation join query involving single interval attribute reduces to multi-way equi-join query involving single real-valued attribute. The project and split operations become identical. The multi-way equi-join query on real-valued data can then be handled in a single map-reduce cycle by projecting all relations. The case of interval data becomes complex because an interval has a finite length. An interval u may start out in a partition-interval p_1, can cross-over to another partition-interval p_2 and overlap with an interval v starting out in partition-interval p_2. Interval v can then may cross-over to another partition-interval p_3 and overlap with an interval w starting-out in partition-interval p_3 and so-on. As all the intervals belonging to an output-tuple may not all share a common point, devising a mechanism for bringing all such intervals at a common reducer becomes a challenging task.

For 2-way interval join it is not a problem as two colocated intervals share a common point. RCCIS gets around this problem by devoting one extra map-reduce cycle to exactly find out which intervals need to be replicated; RCCIS than replicates only such intervals in the second round. *2-way Cascade* and *All-Replicate* are two naive approaches for handling the additional complexity present for case of interval data vis-a-vis real-valued data; while *RCCIS* is an efficient algorithm for handling this additional complexity.

We use the concept of *less-than-order* among relations to efficiently track consistent and crossing interval-sets. Intervals received by a reducer are sorted according to less-than-order. We omit the details due to space constraints.

7. SEQUENCE BASED JOINS

In this section we present an efficient approach for handling multi-way sequence join queries. *2-way Cd* and *All-Rep* are again two naive approaches to handle a sequence based multi-way join query. *RCCIS* does not work for sequence joins as intervals located far apart also satisfy se-

quence predicates and hence each interval needs to be replicated. *RCCIS* hence reduces to *All-Rep*. One hence can not avoid large communication costs in sequence based joins.

However we can significantly improve *All-Rep* by improving on its load balancing aspects among the reducers. *All-Rep* puts a disproportionate burden on some reducers. For example, consider the 2-way query R_1 *before* R_2. *All-Rep* solves it by replicating R_1 and projecting R_2. An output tuple (u, v), $u \in R_1$, $v \in R_2$ is computed by the reducer on which the interval v is projected on. The rightmost reducer hence receives all the intervals of relation R_1 and its load is highest. Figure 4 depicts this. As one moves right, the load on the reducer increases and the load on reducer p_6 is maximum. All intervals of relation R_1 join with R_2 intervals starting in partition-interval p_6 and hence the reducer p_6 runs for a long amount of time while other reducers lie idle. This results in an inefficient use of the resources and hence an inefficient way of solving sequence join. This effect gets more pronounced in case of multi-way sequence join queries as the rightmost reducer receives intervals from all relations except one. Note that *RCCIS* did not have load-balancing issues for colocation joins as *RCCIS* replicates only selected few intervals. We next present and outline the details of *All-Matrix*, which improves upon the load-balancing aspect of *All-Rep*.

7.1 All-Matrix

Basic Idea: As two intervals located far apart satisfy a sequence predicate, solving sequence join can be *likened* to computing a large part of the cross-product of the relations involved. *All-Matrix* divides the computation involved in computing cross-product among individual reducers. This hence requires reducers/interval-data to be visualized as part of higher dimensional space. This allows *All-Matrix* to divide the load of highly loaded reducers in *All-Rep* among multiple reducers while combining the load of lightly loaded reducers in *All-Rep* among fewer reducers; thereby resulting in a better load-balance among reducers vis-a-vis *All-Rep*.

Intuition: Consider the 2-way sequence query R_1 *before* R_2 and Figure 4. The cross-product of R_1 and R_2 is hence in two dimensional space. Consider that relations R_1 and R_2 lie along axis y and x respectively. If both x and y axis are divided in three partition-intervals, then the cross-product space can be visualized as divided among 9 cells. Cell (i, j) receives intervals from relation R_1 which are projected on i^{th} partition across y-axis and of those intervals from relation R_2 which are projected on j^{th} partition across x-axis. The partial computation corresponding to cell (i,j) is assigned to an individual reducer say reducer (i,j). Reducer (i,j) then computes the sequence join among the intervals it receives.

Some of these 9 reducers will not produce any output from the intervals they receive and we call such reducers *inconsistent*. In Figure 4 the inconsistent reducers are shown empty and there are three such reducers. For example, consider the reducer (1,0). The first index belongs to y-axis (relation R_1). It receives intervals u_3 and u_4 from R_1 and v_1 and v_2 from R_2. As u_3 and u_4 start in the 2^{nd} partition while v_1 and v_2 start in 1^{st} partition, none of u_3 or u_4 lie before either of v_1 or v_2. The output of reducer (1,0) hence will be null. Inconsistent reducers can be apriori identified by using the concept of *less-than-order* among relations. Map functions hence avoid communicating any interval to inconsistent reducers.

Note that all 6 reducers receive equal number of intervals and hence the load is balanced. The load is balanced as the load of the reducers receiving high load in *All-Rep* i.e., p_5 and p_6 is distributed across more number of reducers in *All-Matrix* i.e, reducers (0,2), (1,2) and (2,2). Load of reducers with intermediate load in *All-Rep* i.e., p_3 and p_4 is distributed across smaller number of reducers in *All-Matrix* i.e, reducers (0,1) and (1,1). Finally the load of reducers with small load in *All-Rep* i.e, p_1 and p_2 is distributed across only one reducer (0,0) in *All-Matrix*. As the load is balanced, all reducers run for similar amount of time thereby resulting in an efficient use of the resources which leads to a significant improvement vis-a-vis *All-Replicate*.

Details: If there are m relations in the query, *All-Matrix* visualizes the reducers/intervals as part of a m-dimensional space. Consider each axis is divided in o partitions and hence the m-dimensional cross-product space can be seen as a union of o^m cells/reducers. We next define the notion of *consistency* of a reducer.

Consistent Reducer: Consider query $\mathcal{Q}$ and its relation-set $\mathcal{R}$=$\{R_1,R_2,\ldots,R_m\}$. Consider the m-dimensional cross-product space of relations in $\mathcal{R}$, each dimension spanning identical temporal range (say $[t_0, t_n)$). Consider each axis is divided in o equi-sized partitions. A reducer in the m-dimensional matrix is hence identified as an m-tuple $\nabla = (i_1, i_2, \ldots, i_m)$ where $1 \leq i_j \leq o, 1 \leq j \leq m$. A reducer is called consistent if for each condition R_j P R_k in query $\mathcal{Q}$ the following holds:

- If the predicate P enforces R_j to be in less-than-order relationship with R_k, the index i_j is less than or equal to the index i_k and vice-versa.

In Figure 4, a reducer is represented as 2-tuple (i_1, i_2). As *before* predicates enforces R_1 to be less than R_2, the consistent reducers are those with $i_1 \leq i_2$. There are six such reducers out of nine.

Communication Strategy of All-Matrix: If the start-point of an interval u from relation R_k lies in the q^{th} partition-interval along dimension k, map functions communicate u to all reducers ∇=$(i_1, i_2, \ldots, i_m)$ which satisfy the following conditions:

1. **D1:** The reducer ∇ is consistent.
2. **D2:** $i_k = q$.

Computing Output Tuple: Consider an output tuple $\mathcal{U}$=$(u_1,u_2,\ldots,u_m)$, $u_i \in R_i$. Consider intervals in R_k lie along dimension k. The output tuple $\mathcal{U}$ is generated at reducer $(q_1,q_2,\ldots,q_m)$ s.t. q_k is the partition-interval along dimension k in which the interval u_k starts.

Number of MR Cycles: *All-Matrix* computation takes place in a single map-reduce cycle. Map operations read the data and communicate the intervals to reducers according to conditions D1 and D2. Each reducer computes the sequence join for the intervals received. The final output is the union of output computed by all the consistent reducers.

Discussion: In Figure 4, the intervals u_1 and u_2 start in the first partition-interval (i.e, i_1=0) along first dimension (y-axis) and they are communicated to consistent reducers with the first index being 0 i.e. (0,0), (0,1) and (0,2). Intervals v_1 and v_2 start in the first partition-interval (i.e.,

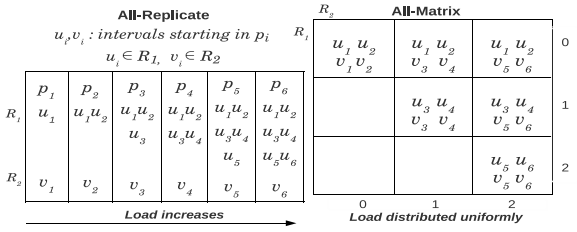
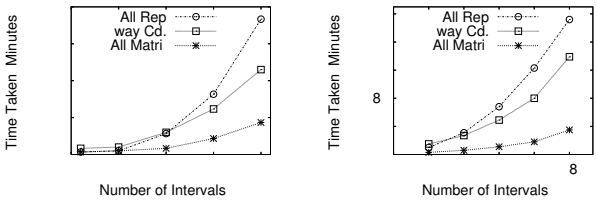


Figure 4: Load Balancing- *All-Rep* vs *All-Matrix*



(a) Synthetic (b) Packet Trace

Figure 5: Performance On Sequence Join

i_2=0) along second dimension (x-axis) and they are communicated to consistent reducers with the second index being 0 i.e., (0,0). Interval-sets $\{u_3, u_4\}$ and $\{u_5, u_6\}$ start in the second and third partition-interval respectively along first dimension and hence they are communicated to consistent reducers with the first index being 1 i.e, [(1,1) and (1,2)] and the first index being 2 i.e., (2,2) respectively. Similarly we can argue about intervals v_3, v_4, v_5 and v_6.

Note that the two conditions D1 and D2 avoid unnecessary communication. An interval is not sent to inconsistent reducers as these reducers won't generate any output. The condition $D2$ ensures that for each output tuple $\mathcal{U}$, exactly one reducer receives all intervals in $\mathcal{U}$. In absence of condition $D2$, an interval will be sent to all consistent reducers. This will result into all consistent reducers receiving all intervals which clearly is highly redundant. For example, consider the output tuple (u_1, v_5) in Figure 4. Only the reducer (0,2) receives both the intervals u_1 and v_5 and computes the output tuple (i.e., q_1=0, q_2=0). If the condition $D2$ is removed both these intervals will be communicated to all six consistent reducers. Similar insights hold for multi-way sequence joins involving more than two relations.

Experimental Evaluation: We consider the 3-way query $\mathcal{Q}_2$=R_1 *before* R_2 and R_2 *before* R_3 and execute *All-Matrix*, *All-Rep* and *2-way Cd*. *All-Matrix* visualizes the reducers as part of 3-dimensional space. We create 6 partitions on all three dimensions and hence 55 reducers are found to be consistent (out of $6^3 = 216$ reducers).

Both 2-way joins in *2-way Cd* (i.e, join of R_1 and R_2 on predicate *before* and the result joining relation R_3) are executed using 2D versions of *All-Matrix* with reducers as part of 2-dimensional cross-product space. For each of these two joins, we create 11 partitions on each dimension. 64 reducers are hence consistent (out of 11^2=121). *All-Rep* is executed using 64 reducers. Partitioning for the three algorithms is chosen in a manner so as the number of consistent reducers are roughly similar (i.e, 55 for All-Matrix, 64 for 2-way Cd and *All-Rep*).

Figure 5(a) presents the results on synthetic data. We vary the size of the relations. The data is generated with temporal range as 0-1000 and the maximum interval length as 100. The distributions dS and dI are taken as uniform. Figure 5(b) presents the results on packet trace P04. Total number of trains in trace P04 are 18K and we randomly sample trains in steps of 3K. Total temporal range for trace P04 is 15 mins. In both the experiments, the approach *All-Matrix* is found to comfortably beat the approaches *2-way Cd* and *All-Rep*. Specifically we note that the large time taken by *All-Rep* is due to lagging reducers. This again highlights the importance of a good load balancing strategy.

7.2 Related Work

The idea of theta-join output space as a cross-product of relations was first used in Okcan et al. [14]. Okcan et al. [14] look at processing 2-way theta join on map-reduce. We extend this idea to multiple dimensions to handle multi-way interval theta join query.

Zhang et al. [17] investigate multi-way theta join query on real-valued data. They handle a multi-way join as a cascade of intermediate chain-joins; each chain join may be 2-way or multi-way. For example, consider the multi-way join query R_1 $P1$ R_2 and R_2 $P2$ R_3 and R_2 $P3$ R_4 where P's are theta join predicates. Zhang et al. [17] may process this join as a cascade of 2 chain joins. First chain join may be multi-way join R_1 $P1$ R_2 and R_2 $P2$ R_3 and the second may be 2-way join which joins the results of first chain join with R_4 on predicate $P3$. For a chain multi-way join query Zhang et al. [17] present a method of communicating data to reduce processes which guarantees minimized volume of data copying over the network as well as evenly distributed workload among reduce tasks. Towards this, the authors present a cost model to estimate the execution time of a map-reduce job. The cost-model requires input data distribution as well as parameters which are system dependent and need to be derived from observations on the execution of real jobs.

Our work is complementary to that of Zhang et al. [17]. Our goal is to present efficient methods for handling the additional complexity introduced due to the presence of interval data vis-a-vis real-valued data. Using the notion of *less-than-order* among relations as implied by Allen's predicates, we identified inconsistent reducers in the multi-dimensional cross-product space. This saves communication cost. We thus improved on the naive way of handling the additional complexity i.e., *All-Replicate*. We can further improve *All-Matrix* by using the cost models and ideas presented in Zhang et al. [17]. The question whether a complex join query should be processed in a single map-reduce job or as multiple jobs is not clear, though the consensus is that processing multi-way join query as a cascade of 2-way is certainly lot worse [17, 8, 10]. We can process multi-way sequence join query as a cascade of multi-way chain joins. However, each of these multi-way chain join query will use only the consistent reducers as defined in this section. Secondly the cost model used in Zhang et al. [17] will need to be updated by taking the distribution of interval lengths into account.

8. HYBRID JOIN QUERIES

In this section, we present how we handle join queries involving single interval attribute and containing both colocation and sequence predicates. We visualize a hybrid query $\mathcal{Q}$ as a join graph $G = (V, E)$. The relations $\mathcal{R}$ form the ver-

tices V. For every join condition R_1 P R_2 in query $\mathcal{Q}$, an edge exists in the graph between the relations R_1 and R_2. Edges are classified as sequence or colocation edges depending upon whether the predicate P is a sequence or colocation predicate.

We next consider the disconnected graph $G^{\square} = (V, E^{\square})$ formed by removing sequence edges from graph G. $G^{\square}$ hence consists of a set of connected components where each component is formed by colocation edges only. If we visualize each connected component as a new relation, the hybrid query $\mathcal{Q}$ can be re-written as a sequence query $\mathcal{Q}^{\square}$ where each sequence predicate is defined over two connected components in $G^{\square}$. Let $\mathcal{R}_c$ denote these new relations. Let $\mathcal{Q}_C$ represent the colocation query encapsulated by the component C.

Consider Figure 6. The hybrid query $\mathcal{Q}_3$ is R_1 *overlaps* R_2 *and* R_2 *overlaps* R_3 *and* R_2 *before* R_4 *and* R_4 *overlaps* R_5. Graph $G^{\square}$ consists of two connected components i.e. R_c={C_1,C_2}. The component C_1 consists of three relations {R_1, R_2, R_3} and the component C_2 consists of two relations {R_4, R_5}. The component C_1 encapsulates the colocation query $\mathcal{Q}_{C_1}$=R_1 *overlaps* R_2 *and* R_2 *overlaps* R_3 and the component C_2 encapsulates the colocation query $\mathcal{Q}_{C_2}$=R_4 *overlaps* R_5. The query $\mathcal{Q}^{\square}$ hence is C_1 *before* C_2.

Two approaches suggest themselves - (1) **First Colocation Then Sequence** (FCTS) and (2) **First Sequence and Then Colocation** (FSTC). FCTS first computes colocation joins using *RCCIS* and then sequence joins using *All-Matrix*. For $\mathcal{Q}_3$ in Figure 6, FCTS first computes queries $\mathcal{Q}_{C_1}$ and $\mathcal{Q}_{C_2}$ using *RCCIS* and then joins the two results using *All-Matrix*. FSTC first executes R_2 *before* R_4 using *All-Matrix* and then the colocation joins using *RCCIS*. Just like *2-way Cd*, both FSTC and FCTS suffer from the problem of joining large intermediate results. We next outline *All-Seq-Matrix* which avoids joining intermediate results.

8.1 All-Seq-Matrix

Consistent Reducer: Let l be the number of connected components in query $\mathcal{Q}^{\square}$ encapsulated by graph $G^{\square}$. *All-Seq-Matrix* hence visualizes the output as the l-dimensional cross-product of l connected-components. A part of this l-dimensional space is assigned to a reducer for processing. Say each dimension is divided in o equi-sized partitions. A reducer is hence represented as l-tuple ∇=(i_1,i_2,...,i_l), $1 \leq i_j \leq o, 1 \leq j \leq l$. A reducer is called consistent if for each condition C_j P C_k in $\mathcal{Q}^{\square}$ the following holds:

- If the predicate P enforces the component C_j to be in less-than-order with C_k, the index i_j is less than or equal to i_k and vice-versa.

Note that the consistency of a reducer is here defined wrt. sequence query $\mathcal{Q}^{\square}$ and not wrt. hybrid query $\mathcal{Q}$. For query $\mathcal{Q}_3$ in Figure 6, there are two components and hence the reducers are visualized as part of 2-dimensional space. Each dimension is divided in three partitions and hence there are 6 consistent reducers (Section 7.1).

Communication to Reducers: We say an interval u belongs to a connected component C if u belongs to any relation R in C. Let dimension i_k belong to connected component C_k. Consider an interval u in relation R which in turn is in the component C_k. Consider that the start-point of interval u lies in the q^{th} partition-interval along dimension i_k. The interval u is communicated to all reducers $\nabla = (i_1, i_2, \ldots, i_l)$ which satisfy the following conditions:

1. **E1:** The reducer ∇ is consistent.

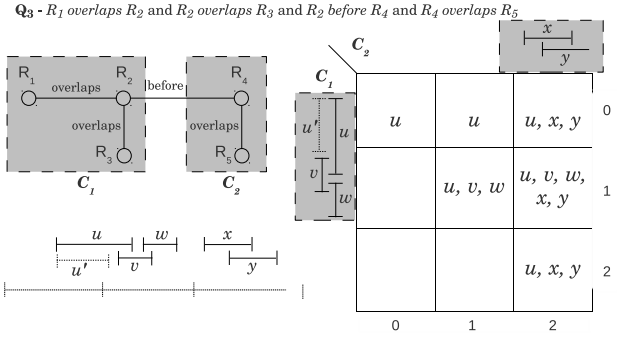


Figure 6:

2. **E2:** If *RCCIS* algorithm would have replicated the interval u while solving the colocation query $\mathcal{Q}_{C_k}$ then $i_k \geq q$ else $i_k = q$.

Computing Output Tuple: Conditions E1 and E2 ensure that all the intervals for an output tuple are received by exactly one reducer. Consider an output tuple $\mathcal{U}$=(u_1, u_2, ..., u_m). Let $\mathcal{U}_{C_k}$ be the intervals in $\mathcal{U}$ which belong to relations in the component C_k. The output tuple $\mathcal{U}$ is generated at the reducer ∇=(q_1,q_2,...,q_l) s.t. q_k is the partition-interval in which the right-most interval in $\mathcal{U}_{C_k}$ starts. Note that the *RCCIS* will project the rightmost interval in $\mathcal{U}_{C_k}$ on the partition-interval q_k while solving the colocation query $\mathcal{Q}_{C_k}$.

Example: Consider Figure 6 and the query $\mathcal{Q}_3$. Intervals u, v, w, x and y belong to relations R_1, R_2, R_3, R_4 and R_5 . Ignore interval $u^{\square}$ at the moment. Intervals in component C_1 (i.e. in relations R_1, R_2, R_3) are visualized as lying across y-axis while the intervals in component C_2 (i.e., in relations R_4, R_5) are visualized as lying across x-axis.

Consider the output tuple $\mathcal{U}$=(u,v,w,x,y). Interval u is sent to all consistent reducers as the interval u is replicated by *RCCIS* while solving the query $\mathcal{Q}_{C_1}$ (Condition E2). Intervals v and w start in the second partition interval i.e. i_1=1. *RCCIS* does not replicate the intervals v and w while solving $\mathcal{Q}_{C_1}$ and hence intervals v and w are communicated to consistent reducers with the first index being 1 i.e., (1,1) and (1,2). Intervals x and y start in the third partition-interval along x-axis i.e., $i_2 = 2$. As *RCCIS* does not replicate the intervals x and y while solving the query $\mathcal{Q}_{C_2}$, intervals x and y are communicated to consistent reducers with the second index being 2 i.e, (0,2), (1,2) and (2,2).

The set $\mathcal{U}_{C_1}$ is {u,v,w} and the set $\mathcal{U}_{C_2}$ is {x,y}. w and y are the right-most intervals in $\mathcal{U}_{C_1}$ and $\mathcal{U}_{C_2}$ and they start-out in the second and third partition-intervals along y and x axis respectively (i.e. i_1=1 and i_2=2). Reducer (1,2) receives all these five intervals and computes the output tuple $\mathcal{U}$. No other reducer receives all the five intervals.

Number of MR Cycles: *All-Seq-Matrix* simultaneously computes and optimizes the sequence query $\mathcal{Q}^{\square}$ using *All-Matrix* and the colocation queries $\mathcal{Q}_{C_k}$'s using *RCCIS*. Unlike FSTC and FCTS, there are no intermediate results. *All-Seq-Matrix* requires two MR cycles. The first MR cycle runs *RCCIS* algorithm to identify which intervals need to be replicated to process colocation queries $\mathcal{Q}_{C_k}$'s. Note that the first MR cycle visualizes reducers as part of 1-dimensional space as it is running *RCCIS*. The second MR cycle communicates intervals to reducers which are visualized as part of l-dimensional space (conditions E1 and E2). Each reducer computes the hybrid join on the intervals it receives and the final output is a union of the output of all reducers.

Table 3: Results for Query $\mathcal{Q}_4$

nI's=(5M, 100K, 1K), dS & dI=Uniform, (t_{min},t_{max})=(0,200K)				
Maximum Interval Length	Time FCTS (hh:mm)	Time *All -Seq-Matrix* (hh:mm)	Time *All-Pruned -Seq-Matrix* (hh:mm)	% intervals pruned in R_1
1000	01:36	00:42	00:29	23.4
800	01:25	00:40	00:29	26.8
600	01:12	00:41	00:25	31.0
400	00:56	00:37	00:21	41.6
200	00:39	00:35	00:16	61.6

Discussion: Lets again consider the scenario where all intervals are of length 0. As discussed in section 6.3, there is no need of *RCCIS* and the condition E2 reduces to $i_k = q$. *All-Seq-Matrix* then runs in a single MR cycle. Additional complexity due to the presence of interval data is handled by applying *RCCIS* algorithm to each colocation sub-join in the hybrid query while at the same time solving the sequence join of the colocation outputs using *All-Matrix*. We next discuss an improvement on *All-Seq-Matrix*.

8.2 Pruned-All-Seq-Matrix (PASM)

Consider the intervals in component C_k. If an interval u s.t. $u \in C_k$ does not appear in the output of colocation query Q_{C_k}, interval u hence naturally won't appear in the output of query $\mathcal{Q}$. For example, consider interval $u^{\square}$ in Figure 6. The interval $u^{\square}$ does not appear in the output of query $\mathcal{Q}_{C_1}$ and hence will not appear in the output of query $\mathcal{Q}_3$.

Such intervals need not be replicated to the reducers as these intervals do not appear in any output tuple. For each component C_k in $G^{\square}$, PASM first identifies intervals in each relation R, $R \in C_k$ which will not appear in the output of query $\mathcal{Q}_{C_k}$. PASM then avoids replicating such intervals. The relations hence can be considered pruned.

This improves the performance on two counts. First the communication cost is reduced as lesser number of intervals are communicated to reducers in *All-Seq-Matrix*. Secondly, each reducer in *All-Seq-Matrix* gets lesser number of intervals to process. The cost of computing partial join-outputs on each reducer is hence reduced. However if the pruning is negligible, this approach may be slightly worse due to the overhead of computing which intervals can be dropped.

PASM hence runs in three MR cycles. First MR cycle runs RCCIS to find out the intervals to replicate. The second MR cycle marks which intervals will not be present in the output of colocation join queries $\mathcal{Q}_{C_k}$'s. The third MR cycle runs *All-Seq-Matrix* on pruned relations i.e, communicates intervals not marked in the second MR cycle to reducers according to conditions E1 and E2.

Experimental Evaluation: We consider the query $\mathcal{Q}_4$ = R_1 *before* R_2 and R_1 *overlaps* R_3. Graph G' hence contains two connected components $\{R_1, R_3\}$ and $\{R_2\}$. *All-Seq-Matrix* hence visualizes the data as 2D matrix. The number of intervals in relations R_1, R_2 and R_3 are fixed at 5M, 100K and 1K respectively. The maximum interval-length (i_{max}) in relation R_3 is varied to study the effect of pruning. The partitioning is identical to as mentioned in Section 7.1.

Table 3 presents the results. As the maximum interval-length is reduced in R_3, lesser and lesser number of intervals in R_1 overlap with any interval in R_3. As a result, more and more intervals in R_1 are pruned. Consequently *Pruned-All-Seq-Matrix* performs better as number of pruned intervals increase. We do not present the numbers for *2-way Cd*, *All-Replicate* due to space constraints. These three approaches were found to be performing significantly worse.

9. MULTI-ATTRIBUTE QUERIES

The three algorithms presented in this paper *RCCIS*, *All-Matrix*, *All-Seq-Matrix* all handle join queries involving single interval attribute. In this section we present *Gen-Matrix* which generalizes *All-Seq-Matrix* algorithm to handle multi-way join query involving multiple interval attributes. Note that as a real-valued attribute can be visualized as an interval of length 0, equality predicate on real-valued attributes as Allen's predicate *equals*, predicates $(<, >)$ on real-valued data as Allen's sequence predicates *before* and *after*; *Gen-Matrix* can handle real-valued attributes as well.

A join condition in such a query $\mathcal{Q}$ is of the form $\langle R, A\rangle$ P $\langle R^{\square}, A^{\square}\rangle$. Here R's, A's and P's denote the relations, attributes and Allen's predicates respectively. As $\mathcal{Q}$ contains multiple attributes, a relation R may have more than one attribute involved in query $\mathcal{Q}$ and hence the join conditions contain the pair $\langle R, A\rangle$ (and not just the relation R).

Consider the query as join graph $G = (V, E)$. Relation-attribute pairs $\langle R, A\rangle$ form the vertices V. For each join condition $\langle R,A\rangle$ P $\langle R^{\square},A^{\square}\rangle$ in query $\mathcal{Q}$, an edge exists in the graph between vertices $\langle R,A\rangle$ and $\langle R^{\square}, A^{\square}\rangle$. The edge is classified as sequence or colocation depending on predicate P. By dropping sequence edges in G, we get the disconnected graph $G^{\square}$. Note that unlike in *All-Seq-Matrix*, the graph G may already be a disconnected graph due to the presence of more than one attribute. We again visualize each disconnected component in $G^{\square}$ as a new relation. Let R_c denote these new relations (connected-components). Let Q_C represent the colocation query encapsulated by component C.

Consider the query $\mathcal{Q}_5$ = $R_1.I$ *before* $R_2.I$ and $R_1.I$ *Overlaps* $R_3.I$ and $R_1.A$=$R_3.A$ and $R_2.B$=$R_3.B$. Here I is an interval attribute while A and B are real-valued attributes. Graph $G^{\square}$ for query $\mathcal{Q}_5$ consists of four connected components: C_1=$\{\langle R_1,I\rangle,\langle R_3,I\rangle\}$, C_2=$\{\langle R_2,I\rangle\}$, C_3=$\{\langle R_1, A\rangle, \langle R_3, A\rangle\}$ and C_4=$\{\langle R_2,B\ \rangle ,\langle R_3,B\ \rangle\}$.

Less-Than Order between two connected components: We say a connected component C_i is in *less-than* order with component C_j if there exists a join condition $\langle R, A\rangle$ P $\langle R^{\square}, A^{\square}\rangle$ or $\langle R^{\square},A^{\square}\rangle$ P $\langle R,A\rangle$ in $\mathcal{Q}$ s.t. $\langle R, A\rangle \in C_i$, $\langle R^{\square}, A^{\square}\rangle \in C_j$ and P_k is a sequence predicate which enforces a less-than order between $\langle R,A\rangle$ and $\langle\ R, A\rangle$.

Note that if there exist more than one such join conditions then all such join predicates must enforce the same less-than order between the two components. Otherwise, no set of tuples can satisfy all predicates of query $\mathcal{Q}$ and hence the output of query $\mathcal{Q}$ will be null. We next present the algorithm *Gen-Matrix* in detail.

9.1 Gen-Matrix

Consistent Reducer: If there are l connected components in $G^{\square}$, *Gen-Matrix* visualizes the join output as l-dimensional space formed by cross-product of l connected components. Say each dimension is divided into o equi-sized partitions. A reducer ∇ is hence represented as an l-tuple $(i_1,i_2,\ldots,i_l)$, $1 \le i_j \le o, 1 \le j \le l$. The dimension i_j belongs to j^{th} component. A reducer ∇ is consistent if for all j and k s.t. if C_j is less-than C_k, i_j is less-than i_k. We now present the approach *Gen-Matrix* in detail.

Communication to Reducers: As each dimension in the l-dimensional space is divided into o partition-intervals, there are o^l cells and hence o^l reducers. Consider a relation R in $\mathcal{R}$. Let $\mathcal{A}$ be the join attributes in R. For each attribute

A in $\mathcal{A}$, *Gen-Matrix* communicates a tuple r from R to the reducers as outlined below. Let interval a represent the value of attribute A in R. Consider that the interval a starts in q^{th} partition-interval. Let C_k be the connected component in R_c in which the vertex $\langle R,A\rangle$ lies. The tuple r is routed to all reducers ∇ which satisfy the following two conditions:

1. **E1:** The reducer ∇ is consistent.
2. **E2:** If *RCCIS* algorithm would have replicated the interval a while solving the colocation query $\mathcal{Q}_{C_k}$ then $i_k \geq q$ else $i_k = q$.

Computing Output Tuple: Consider an output tuple $\mathcal{T}=(r_1,r_2,\ldots,r_m)$ where r_i is a tuple in R_i. Let $\mathcal{T}_{C_k}$ be the tuples in $\mathcal{T}$ which belong to relations in the connected component C_k. The output tuple $\mathcal{T}$ is generated at the reducer $\nabla=(q_1,q_2,\ldots,q_l)$ where the indices q_k satisfy the following:

- Let $\mathcal{U}$ be the set of intervals; each interval in $\mathcal{U}$ is the value of interval attribute A in tuple r from relation R s.t. the pair $\langle$R, A$\rangle$ forms a vertex in the connected component C_k and $r \in \mathcal{T}_{C_k}$. q_k then is the partition-interval in which the right-most interval in $\mathcal{U}$ starts.

Experimental Evaluation: Consider the query $\mathcal{Q}_5$ again. We execute *Gen-Matrix* on synthetic data with 375 reducers. Table 4 presents the results. Here we vary the size of the three relations. *Gen-Matrix* on $\mathcal{Q}_5$ requires four dimensions and hence the data is visualized as part of a 4-dimensional space. Query $\mathcal{Q}_5$ only enforces a less-than order between C_1 and C_2. Each dimension is partitioned into 5 intervals and hence 375 (out of 625) reducers are found to be consistent. As the size of the three relations are increased in fixed steps, the time taken by *Gen-Matrix* increases linearly.

Table 4: Gen-Matrix

dI, dS = Uniform, dA, dB = Uniform (t_{min},t_{max})=(0,100K), (i_{min},i_{max})=(1, 1000)	
nI's	Time (mm:ss)
100K, 10K, 100K	11:34
110K, 11K, 110K	14:09
120K, 12K, 120K	17:28
130K, 13K, 130K	18:10
140K, 14K, 140K	22:19

9.2 Discussion and Related Work

Gen-Matrix extends *All-Seq-Matrix* to handle multiple attribute. To start with *RCCIS* optimizes the computation of colocation joins by exploiting the property that only nearby intervals satisfy colocation predicates. As two intervals located far apart satisfy sequence predicates, *All-Matrix* divides the computation in the cross-product space to carry out a better load-balancing. *All-Seq-Matrix* and *Gen-Matrix* use multi-dimensional space to load-balance for sequence sub-joins while locally optimizing colocation sub-joins using *RCCIS*. The only difference being that all *RCCIS* instances are working on same interval attribute in *All-Seq-Matrix* but on different attributes in *Gen-Matrix*.

There are two main related works. Gupta et al. [8] present an algorithm to process multi-way spatial colocation joins on map-reduce where each spatial object is a rectangle. As mentioned in Section 1, a rectangle can be defined as a set of two intervals - length and breadth. The algorithms presented in Gupta et al. [8] are an expansion of a special case of *Gen-Matrix* algorithm, where the queries involve (a) only two interval attributes, (b) only *overlap* predicates and (c) each relation consists of these two interval attributes. There are no sequence predicates and there is no notion of *less-than-order* among relations. In this paper we have discussed the full spectrum of interval join queries and presented four different algorithm to handle four different classes of queries.

Afrati et al. [2] present an algorithm for partitioning the cross-product space for multi-way equi joins on real-valued data. For the special case of multi-way colocation joins on interval data, their approach can be integrated with *RCCIS* algorithm to improve *Gen-Matrix*. As discussed in this paper, the algorithms for real-valued data can not be directly used on interval data due to intervals having a finite length. Our work is hence complementary to that of Afrati et al. [2].

10. RELATED WORK

Related work is mentioned across sections 1, 7.2 and 9.2. We also contrast the algorithms presented in this paper with the related work in these sections.

11. CONCLUSIONS

In this paper we carried out a comprehensive investigation of multi-way interval join queries on map-reduce platform. We developed four different algorithms to handle different classes of interval join queries and carried out an experimental evaluation to showcase the efficacy of these algorithms. We also discussed how interval join queries pose multiple challenges vis-a-vis join queries on real-valued data. We next plan to integrate the ideas developed in Afrati et al [2] and Zhang et al. [17] with the algorithms presented in this paper. We also plan to explore more avenues for analyzing interval data on map-reduce e.g., temporal pattern mining etc.

12. REFERENCES

[1] MAWI working group traffic archive http://tracer.csl.sony.co.jp/mawi.
[2] F. N. Afrati and et al. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.
[3] J. Allen. Maintaining knowledge about temporal intervals. *Communications of ACM*, 26(11), 1983.
[4] J. B. Buck and et al. SciHadoop: Array Based Query Processing In Hadoop. In *SC*, 2011.
[5] J. Dean and et al. MapReduce : Simplified data processing on large clusters. *Comm. of ACM*, 51(1), 2008.
[6] A. Eldway and et al. A Demonstration of SpatialHadoop: An Efficient MapReduce framework for Spatial Data. In *PVLDB*, 2013.
[7] A. Ghoting and et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.
[8] H. Gupta and et al. Processing Multi-way Spatial Joins On Map-Reduce. In *EDBT*, 2013.
[9] R. Jain and et al. Packet trains measurement and a new model for computer network traffic. *IEEE journal on selected areas in Communications*, 4(6), 1986.
[10] R. Lee and et al. YSMART: Yet another sql-to-mapreduce translator. In *ICDCS*, 2011.
[11] W. Lu. and et al. Efficient processing of K-NN joins using Map-Reduce. In *VLDB*, 2012.
[12] Q. Ma and et al. Query Processing of Massive Trajectory Data Based on Map-Reduce. In *CloudDB*, 2009.
[13] A. Metwally and et al. V-SMART-JOIN: A scalable Map-Reduce framework for all-pair similarity joins. In *VLDB*, 2012.
[14] A. Okcan and et al. Processing theta-joins using MapReduce. In *SIGMOD*, 2011.
[15] H. Tan and et al.i. CloST: A Hadoop-based storage system for big spatio-temporal data analytics. In *CIKM*, 2012.
[16] R. Vernica and et al. Efficient parallel set-similarity joins using map-reduce. In *SIGMOD*, 2010.
[17] X. Zhang and et al. Efficient Multi-way Theta Join Processing Using MapReduce. In *VLDB*, 2012.