Processing Multi-Way Spatial Joins on Map-Reduce

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

ABSTRACT

In this paper we investigate the problem of processing multi-way spatial joins on map-reduce platform. We look at two common spatial predicates - *overlap* and *range*. We address these two classes of join queries, discuss the challenges and outline novel approaches for executing these queries on a map-reduce framework. We then discuss how we can process join queries involving both *overlap* and *range* predicates. Specifically we present a *Controlled-Replicate* framework using which we design the approaches presented in this paper. The *Controlled-Replicate* framework is carefully engineered to minimize the communication among cluster nodes. Through experimental evaluations we discuss the complexity of the problem under investigation, details of *Controlled-Replicate* framework and demonstrate that the proposed approaches comfortably outperform naive approaches.

1. INTRODUCTION

Spatial data consists of points, lines, rectangles, polygons and more complex objects composed from simple ones. Spatial data naturally arises in multiple domains e.g., satellite images, digital video, multi-media documents, medical information systems, robotics etc. Increasing availability of such data has rendered spatial query processing as one of the most active research areas in the database community. One of the most important spatial queries is spatial join which retrieves from datasets all object pairs that satisfy a certain spatial predicate [14]. For example the query 'find all cities adjacent to a forest and overlap with a river' involves a spatial join between *city*, *forest* and *river* datasets. The query 'Find all objects within 10 m of each other that have identical colors but different brightness' involves a spatial self-join between *object* datasets.

Many studies in the literature have looked at how to optimize spatial joins in a database engine. These include both optimizing 2-way spatial joins [5, 7, 6, 11, 12, 18, 19, 10] as well as optimizing multi-way joins [17, 15]. In this paper we investigate the problem of optimizing multi-way spatial joins on map-reduce platform. Map-reduce [8] is a framework for parallel processing proposed by Google for large scale data processing and many modern applications use map-reduce as their data-processing platform. This has inspired many studies to investigate efficient approaches for handling join queries on map-reduce platform [16, 3]. Few studies have also looked at how to compute 2-way spatial joins on map-reduce platform [21, 23, 22]. However the processing of multi-way spatial joins on map-reduce has not received much attention. To the best of our knowledge, this is the first study to do so.

Implementation of a map-reduce program entails writing of map and reduce functions. A map function reads the input and converts the input to an intermediate form consisting of a set of key-value pairs. These key-value pairs are collected by the map-reduce engine and are routed to reducers in a manner that all pairs with identical key are routed to a single reducer. Efficiency of a map-reduce program often hinges upon the number of intermediate key-value pairs being generated. Larger the number of intermediate key-value pairs, higher the communication cost. Hence a map-reduce algorithm should be designed in a manner such that it produces minimal number of intermediate key-value pairs. The approaches proposed in this paper are developed with an objective of minimizing communication cost.

Contributions: In this paper we design novel algorithms for handling multi-way spatial join queries on map-reduce platform. Specifically we look at two common spatial predicates - *overlap* and *range* on datasets consisting of rectangular objects. The *overlap* predicate looks for the pairs of rectangles which intersect with each other while the *range* predicate looks for pairs of rectangles within a certain distance of each other. We develop a *Controlled-Replicate* framework coupled with *project-split-replicate* notation using which we handle multi-way spatial join queries. This framework outlines how rectangles are communicated to different reducers. We first present novel approaches to handle multi-way *overlap* and *range* join queries and discuss the challenges arising therein. Finally we present how we can handle general spatial multi-way join queries which involve both *overlap* and *range* predicates. By carrying out an experimental study over synthetic as well as real-life data we show that the approaches presented in the paper comfortably beat the naive approaches.

1.1 Object Model - Why only Rectangles?

Representing spatial objects with their minimum bounding rectangle (MBR) is a standard approach in spatial join

EDBT/ICDT '13 March 18 - 22 2013, Genoa, Italy

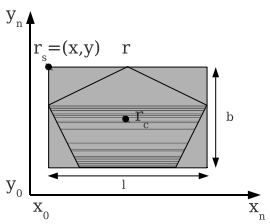


Figure 1: Representation of a MBR

literature [15]. A spatial object may be circular or may have multiple vertices, may be concave or convex etc and hence the representation of a spatial object can be large and complex. Finding out whether two spatial objects satisfy a certain predicate or not, can be computationally very expensive. Spatial joins are hence carried out in two steps: *filter* and *refinement*. Each spatial object is approximated using its minimum bounding rectangle (MBR). In the *filter* step, instead of the two spatial objects, their MBRs are checked whether they satisfy a spatial predicate or not. Checking whether two MBRs satisfy a certain predicate is computationally inexpensive as the representation of an MBR is simple. If the MBRs do not satisfy the predicate, then the two objects will not satisfy the predicate; however the reverse is not true. The *filter* step hence produces a super-set of the actual result. For each pair of MBRs output by the *filter* step, the *refinement* step then checks whether the two objects actually satisfy the predicate or not. Computationally expensive geometric procedures hence are employed in the *refinement* step. As a result the two-step process is computationally more efficient as compared to checking every pair of objects whether they satisfy a spatial predicate or not.

The objective of the approaches developed in this paper is to hence efficiently carry out the *filter* step on MBRs. A *refine* operation is carried out in the end to check whether the objects actually satisfy the spatial multi-way join or not.

A relation is hence visualized as consisting of a set of rectangles and the spatial multi-way join query is evaluated over the MBRs. Figure 1 shows a pentagon object and its MBR. A rectangle r is represented as (x, y, l, b) where x and y represent the coordinates of top-left vertex of the rectangle; while l and b represent the length and breadth of the rectangle. We also refer to the top-left vertex as the start-point of the rectangle. The x-range and y-range are $[x_0, x_n]$ and $[y_0, y_n]$ respectively i.e. all MBRs lie within the 2D space defined by these two ranges.

1.2 Query Model

In this paper, we consider two spatial predicates - *overlap* and *range* defined as follows:

Overlap(r_1, r_2): true if rectangles r_1 overlaps with rectangle r_2

Range(r_1, r_2, d): true if the distance between any two points of rectangles r_1 and r_2 is less than d

Overlap(R_1, R_2) finds the pairs of rectangles (r_1, r_2), $r_1 \in R_1$, $r_2 \in R_2$ and r_1 overlaps with r_2. *Range*(R_1, R_2, d) finds the pairs of rectangles (r_1, r_2), $r_1 \in R_1$, $r_2 \in R_2$ s.t. at least one point of rectangle r_1 is within distance d of at least one point in r_2. Two objects can overlap only if their MBRs overlap and two objects can be within distance d only of their MBRs are within distance d.

We denote a multi-way spatial query as a conjunction of triples $T_i = (P_i, R_{i,1}, R_{i,2})$ as follows:

$$\mathcal{Q} = \{(P_1, R_{1,1}, R_{1,2}) \wedge (P_2, R_{2,1}, R_{2,2}) \wedge \ldots \wedge (P_n, R_{n,1}, R_{n,2})\} \quad (1)$$

Here P_i's are spatial predicates while $R_{i,1}$ and $R_{i,2}$ represent the two relations being joined on spatial predicate P_i.The query $\mathcal{Q}$ is an overlap or range query if all the predicates P_i's are either *overlap* or *range*. Overlap and range predicates are also represented using Ov and $Ra(d)$ respectively.

We visualize the query as a join graph $G(V, E)$ with each relation as a vertex in the graph and for every triple (P_i, $R_{i,1}$, $R_{i,2}$) in query $\mathcal{Q}$ there is an edge between the vertices for relations $R_{i,1}$ and $R_{i,2}$ with the edge weight being 0 if predicate P_i is an *overlap* predicate and d if P_i is a range predicate with distance parameter d.

Organization: Section 2 presents the basics of map-reduce programming model. Section 3 discusses the related work. Section 4 develops the *project-split-replicate* notation which is used in rest of the paper. Section 5 discusses 2-way spatial joins in terms of *project-split-replicate* notation. Section 6 presents two naive approaches *2-way Cascade* and *All-Replicate* for handling multi-way spatial joins and discusses why these approaches are naive. Section 7 presents *Controlled-Replicate* framework and discusses how we can handle multi-way *overlap* join queries. Section 8 proposes *Controlled-Replicate* based approaches for handling multi-way spatial *range* join queries. Section 9 discusses how we handle general spatial join queries which involve both *overlap* and *range* predicates. Section 7, 8 and 9 also present the experimental evaluation on the approaches presented in the respective sections. Section 10 concludes the paper with a mention of future work.

2. BACKGROUND: MAP-REDUCE

A map-reduce program consists of two user-defined functions map and reduce. The signatures of these two functions are as follows:

map: (k_1,v_1)→[k_2,v_2]
reduce:(k_2,[v_2])→[k_3,v_3]

Every input record is parsed as a key-value pair (k_1,v_1). Map function applies a user defined logic on each input key-value pair (k_1,v_1) and transforms it into a set of intermediate key-value pairs ([k_2,v_2]). These map outputs are collected and the keys of type k_2 are assigned to reducer nodes. Then the reduce function applies a user-defined logic to all intermediate values [v_2] associated with the same k_2 and produces a list of final output key-value pairs [k_3,v_3].

The map-reduce framework parallelizes the map and reduce operations by dividing the responsibilities of these operations among many nodes. Input data is distributed across several physical locations on a distributed file system (DFS). On initialization of a map-reduce job, the input data is partitioned among multiple mapper nodes, which are tasks responsible for applying the map function. Mappers fetch the input data from DFS and write the set of intermediate key-value pairs to a set of files. Reducers then access these files and compute the final output. Once a reducer has received its files from all mappers, it merges and sorts the files by keys and reduces each key in turn, outputting the resulting tuples to files on the distributed file system.

3. RELATED WORK

Spatial join processing on RDBMS is a well developed research area. A number of studies have investigated optimization of both 2-way and multi-way spatial joins in an RDBMS [5, 7, 6, 11, 12, 17, 15]. In this paper we study the related problem of optimizing multi-way spatial joins on map-reduce platform.

Few studies have recently looked at optimizing 2-way spatial join processing on map-reduce [23, 21, 22]. In addition these studies also look at other spatial queries like kNN - finding k objects in a dataset that are nearest to a query point q, ANN - for an object in dataset 1 find its nearest neighbor in dataset 2, containment - find objects which contain a point etc. However none of these studies consider multi-way spatial join processing (or any other query processing). Ours is the first study to analyze the problem in detail. The basic approach these studies employ to handle 2-way spatial queries is to visualize the 2D space as a grid of reducers and let each reducer handle a different part of the space. We also employ a similar approach to handle multi-way spatial joins on map-reduce.

Optimization techniques for the evaluation of join queries on map-reduce platform has seen a flurry of activity last few years. Blanas et al. [4] look at various algorithms for optimizing 2-way equi-joins on map-reduce. Afrati et al. [3] look at multiway equi-joins on Map-Reduce platform. Okcan et al. [16] look at processing 2-way inequality joins on map-reduce platform. One common objective in all these studies is to minimize the communication cost among the cluster nodes as well as ensure that the reducers are load-balanced. We also design our solutions for handling multi-way spatial joins with a similar objective by exploiting various properties which arise due to relative spatial location of objects.

Few studies recently have looked at optimizing join queries for specialized scenarios. Lu et al. [13] look at optimizing K-NN joins on map-reduce platform. Vernica et al. [20] look at optimizing set-similarity joins on map-reduce. In this paper we look at multi-way join processing on spatial datasets which has not been studied previously.

4. PROJECT-SPLIT-REPLICATE

Partitioning: Let the complete x-range and y-range be [x_0,x_n) and [y_0,y_n) i.e., all rectangles lie within this space. A rectilinear *partitioning* divides the space into a set of disjoint rectangles which taken together cover the whole space. We equivalently call these rectangles as partition-cells. Partition-cells in each row have the same breadth and partition-cells in each column have the same length. Figure 2(a) shows an example of partitioning. Here the whole space is partitioned into 16 cells.

We represent a partitioning as $\mathcal{C}$=(c_1,c_2,..., c_q) where the c_j's represent the individual partition-cells and q is the total number of partition-cells in $\mathcal{C}$. We also denote each partition-cell by their indices (i, j). We will be using these two notations interchangeably in this paper. 16 cells in Figure 2(a) are denoted by their ids i.e. (1, 2, ..., 16) or their indices ((1,1), (1,2), ... , (4,4)).

Distance between a rectangle and partition-cell: The distance between a partition-cell c and a rectangle r is defined as the minimum distance between any point p_1 in partition-cell c and any point p_2 within the rectangle. Mathematically it can be written down as follows:

$$dist(c,r) = \min_{p_1,p_2} dist(p_1,p_2), \forall (p_1,p_2) p_1 \in c, p_2 \in r \quad (2)$$

Cell of a Rectangle: Given a rectangle u, its cell c_u is defined as the partition-cell in which the start-point of rectangle u lies. In Figure 2(a), the cell of rectangle r_1 (i.e., c_{r_1}) is 6 as the start point of rectangle r_1 lies in cell 6. Similarly the cell of rectangle r_2, (i.e., c_{r_2}) is 3.

Cells in the 4^{th} Quadrant wrt. a Rectangle: In this paper we will be heavily using this notion. Consider a rectangle u and its cell c_u. If we divide the whole 2D space by taking the start-point of cell c_u as origin, than the cells lying in the fourth quadrant are said to be the cells in the 4^{th} quadrant wrt. rectangle u and denoted as $\mathcal{C}_4(u)$. Mathematically it is defined as follows:

$$\mathcal{C}_4(u) = \{c_i\} \text{ s.t. } c_i.x \geq c_u.x \ \& \ c_i.y \leq c_u.y$$

In Figure 2(a), cells 6-8, 10-12 and 14-16 are in the 4^{th} quadrant wrt. rectangle r_1. Cell of rectangle r_1 i.e., c_{r_1} is 6 and the four quadrants (marked by Q_1, Q_2, Q_3 and Q_4) formed by division of 2D space by taking start-point of cell 6 as origin are also shown in Figure 2(a).

We next define the following three operations *project*, *split* and *replicate*. Collectively we call these three operations as transform operations. These operations transform a rectangle wrt. a partitioning in three different manners. Intermediate key-value pairs in the approaches we develop in this paper are generated using these transform operations. The output consists of key-value pairs where key is the partition-cell id and the value is the input rectangle.

Project: Project operation returns the cell of a rectangle i.e., it determines the partition-cell in which the start-point of the rectangle lies. The projection of a rectangle u on a partitioning $\mathcal{C}$ results in the generation of a single key-value pair (c_u, u) where c_u is the partition-cell within which the start-point of the rectangle u lies.

Project(u, $\mathcal{C}$)$\rightarrow$ (c_u,u)

Split: Split operation determines all the partition-cells which have at-least one point in common with the rectangle. Consequently for each such partition-cell, a key-value pair is generated and hence a set of key-value pairs is returned for each rectangle.

Split(u, $\mathcal{C}$)$\rightarrow$ $\{(c_i,u)\}$, $\forall(i)$ s.t. $u \cap c_i \neq \phi$

Replicate: The replicate operation returns all partition cells which satisfy a certain condition.

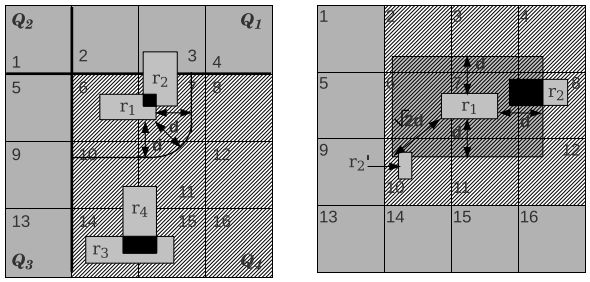
Replicate(u, $\mathcal{C}$, f)$\rightarrow$ $\{(c_i,u)\}$, $\forall(i)$ $f(c_i,u)$=true

In this paper we will be mainly considering two functions f_1 and f_2 as described next. The function f_1 replicates the rectangle u to all partition-cells which lie in the fourth quadrant wrt. rectangle u. The function f_2 takes an additional parameter d and replicates the rectangle u to all cells which (1) lie in the fourth quadrant wrt. rectangle u and (2) are within distance d of the rectangle u.

Mathematically, these two functions are written down as follows.

$$f_1(c,u) = c.x \geq c_u.x \ \& \ c.y \leq c_u.y$$

$$f_2(c,u,d) = f_1(c,u) \ \& \ dist(c,u) \leq d$$



Project(r_1): [{6,r_1}] Split(r_1):[{6,r_1},{7,r_1}]
Replicate(r_1,f_1): [{6,r_1},{7,r_1},{8,r_1},{10,r_1},{11,r_1}, {12,r_1},{14,r_1},{15,r_1},{16,r_1},]
Replicate(r_1,f_2): [{6,r_1},{7,r_1},{10,r_1},{11,r_1}]

(c)

Figure 2: Project, Split and Replicate Example

Figure 2(c) presents the output of project, split and replicate operations on rectangle r_1 in Figure 2(a). The project operation returns cell 6 as cell 6 contains the start-point of rectangle r_1. The split operation returns the cells 6 and 7 as only these two cells intersect with rectangle r_1. Replicate operation with function f_1 returns cells in 4^{th} quadrant wrt. rectangle r_1 i.e., cells 6-8, 10-12, 14-16. Replicate operation with function f_2 returns cells 6,7,10 and 11 as only these cells are within distance d of rectangle r_1. The shaded cells represent the output of replicate operations with function f_1 in Figure 2(a).

Projecting, Splitting and Replicating a relation: Equivalently we define projecting (Project(R,$\mathcal{C}$)) splitting (Split(R,$\mathcal{C}$)) and replicating (Replicate(R,$\mathcal{C}$)) a relation R wrt. a partitioning $\mathcal{C}$ as follows:

Project(R, $\mathcal{C}$): $\forall(j)$ Project(u_j, $\mathcal{C}$) s.t. $u_j \in R$
Split(R, $\mathcal{C}$): $\forall(j)$ Split(u_j, $\mathcal{C}$) s.t. $u_j \in R$
Replicate(R, $\mathcal{C}$): $\forall(j)$ Replicate(u_j, $\mathcal{C}$) s.t. $u_j \in R$

Project(R, $\mathcal{C}$) represents the output of project operation on each rectangle in R. The size of this operation hence equals the size of relation R. *Split(R, $\mathcal{C}$)* represents the output of split operation on each rectangle in R and similarly *Replicate(R, $\mathcal{C}$)* represents the output of replicate operation on each rectangle in R.

Notation: To make the notation simpler, we omit the partitioning parameter $\mathcal{C}$ while talking about any of *project*, *split* or *replication* operation. In the rest of the paper we may avoid mentioning the replication function f_1 and f_2 explicitly. From the context it will be clear which function is implied. Table 1 summarizes the symbols used in this paper.

5. 2-WAY SPATIAL JOINS

Binary joins have been studied in the literature before [23, 22]. In this section we first outline the same solutions using the *project-split-replicate* notation. Secondly understanding how 2-way spatial joins can be computed on map-reduce platform is essential to understand how multi-way joins need to be carried out.

5.1 2-way Join Algorithm Blueprint

Let the number of reducers be k and let us denote these reducers by ids 1,2,...,k. We first divide the 2D space into a grid containing k partition-cells (i.e., we divide x and y axis in $\sqrt{k}$ partitions each). We either project, split or replicate the relations (depending on the exact predicate involved). Intermediate key-value pairs generated as a result of these operations are routed to the reducers. An intermediate key-value pair (c_i, u) is routed to the reducer c_i. All pairs with partition-cell c_i as key are hence processed by one reducer only i.e., c_i.

After all the intermediate key-value pairs have been communicated to the relevant reducers, each reducer has got all the information to generate a part of the output. If two rectangles from two relations agree on the predicate then these two rectangles must be present at, at least one of the reducers. Each reducer computes a part of the spatial join output. Combining the output of all the reducers produces the complete join output.

Notation: As one reducer handles the key-value pairs containing only one partition-cell, we also denote the reducers with the same ids i.e., c_i. We will be using the terms 'reducer' and 'cell' loosely and interchangeably. We will not be using the exact phrases 'reducer corresponding to the cell' and 'cell corresponding to the reducer'. It will be clear from the context what is implied.

5.2 Overlap Predicate

We next look at the *Overlap* predicate. Let R_1 and R_2 be the two relations being joined. We can process the *overlap* join by splitting the two relations. Let $r_1 \in R_1$ and $r_2 \in R_2$ be two rectangles which satisfy the *Overlap* predicate. Hence there must be at least one reducer which will receive both r_1 and r_2. Such reducers can output that the rectangles r_1 and r_2 overlap. As there can be more than one reducers which receive both r_1 and r_2, we need to remove the duplicates from the output. The final output is hence scanned once again and the duplicates are removed. Alternatively we can adopt a duplicate avoidance mechanism so that we need not remove duplicates from the final output. A duplicate avoidance mechanism ensures that one output tuple is generated only at one of the reducer. We adopt the following duplicate avoidance mechanism [9].

We first compute the area overlapping between the two rectangles r_1 and r_2. This area will also be a rectangle and let's denote this by r_{o12}. Which-ever cell contains the start-point of the rectangle r_{o12}, computes the output tuple (r_1, r_2). As only one cell can contain the start-point of the rectangle r_{o12}, there will be no duplicate.

Consider the rectangles r_3 ($\in R_1$) and r_4 ($\in R_2$) in Figure 2(a). x and y axis are divided in 4 partitions and hence there are 16 reducers in play. The overlapped area between rectangles r_3 and r_4 is shown black. Reducers 14 and 15 will receive both the rectangles r_3 and r_4. However the start-point of overlapping area lies in cell 14. The output tuple (r_3,r_4) is hence computed by reducer 14.

Note that an *overlap* join can not be computed by projecting one relation and splitting the other. Consider that the relation R_1 is projected and relation R_2 is split. For a counter-example, consider the rectangles r_1 ($\in R_1$) and r_2 ($\in R_2$) in Figure 2(a). As rectangle r_1 is projected, only reducer 6 receives rectangle r_1. As rectangle r_2 is split, reducers 3 and 7 receive the rectangle r_2. Hence, no reducer receives both the rectangles r_1 and r_2; and hence this output tuple will not be computed.

Table 1: Terminology and Notation

R_1, R_2, R_3	Relations
u, v, w, x r_1, r_2, r_3	Rectangles
p_1, p_2	A point
$\mathcal{Q}$	A Spatial Join Query
$\mathcal{R}$	A set of relations, Set of Relations in query $\mathcal{Q}$
$\mathcal{U}$	A set of rectangles with each rectangle belonging to a different relation
P	A spatial predicate
c	partition-cell
$\mathcal{C}$	A partitioning, A set of partition-cells
c_u, c_r	Cell in which rectangle u (r) starts

5.3 Range Predicate

Let R_1 and R_2 be the two relations being joined and let d be the distance parameter of the *range* predicate. A range predicate computes the pairs of rectangles which are within distance d of each-other. We first define the concept of enlarging a rectangle by d units.

Enlarging a rectangle by d units: Let (x_1, y_1) be the top-left vertex of a rectangle and let (x_2, y_2) be the bottom-right vertex . The top-left vertex of the enlarged rectangle is given by $(x_1 - d, y_1 + d)$ and the bottom-right vertex is given by $(x_2 + d, y_2 - d)$. Consider the Figure 2(b). Rectangle r_1 is shown and its enlarged rectangle is also shown. Let us denote the rectangle obtained by enlarging rectangle r by d units as $r^e(d)$.

We can process the *Range* predicate by splitting the relation R_2 and communicating a rectangle r_1 in R_1 to reducers corresponding to all partition-cells which overlap with the rectangle obtained by enlarging r_1 by d units (i.e, $r_1^e(d)$). For example rectangle r_1 in Figure 2(b) is communicated to reducers 2-4, 6-8 and 10-12 as these cells overlap with the enlarged rectangle of r_1. Rectangle r_2 is communicated to reducers 7 and 8 as these two cells overlap with rectangle r_2.

There can be more than one reducers which will receive both the rectangles r_1 and r_2. We hence need to have a duplicate avoidance mechanism. We once again compute the rectangular area overlapping between $r_1^e(d)$ and r_2. The cell which contains the start-point of this rectangular area computes the output tuple (r_1, r_2) if r_1 and r_2 are within distance d.

In Figure 2(b), reducers 7 and 8 receive both rectangle r_1 and r_2 and reducer 7 computes the output tuple (r_1, r_2) as the start-point of the overlapping area between r_1^e and r_2 lies in cell 7.

Note that we still need to check whether rectangle r_1 and r_2 are within distance d as the overlap between enlarged rectangle of r_1 (i.e., r_1^e) and r_2 does not guarantee that r_1 and r_2 are within distance d. For a counter-example, consider rectangles r_1 and $r_2^{'}$ in Figure 2(b). Rectangle $r_2^{'}$ overlaps with r_1^e but r_1 and $r_2^{'}$ are more than distance d apart.

If r_1 and r_2 are within distance d then r_2 will overlap with enlarged rectangle $r_1^e(d)$ but the reverse is not true. Hence it suffices to split r_2 and replicate r_1 to reducers which overlap with $r_1^e(d)$.

6. MULTIWAY JOINS - NAIVE METHODS

In this section we present two naive approaches to solve multi-way spatial join queries and discuss why these approaches are naive.

6.1 Naive Approaches

- ***2-way Cascade*:** This approach processes a multi-way join query as a series of 2-way joins. Each 2-way join is handled as discussed in Section 5. For example consider the query $\mathcal{Q}_1$= R_1 *Overlaps* R_2 and R_2 *Overlaps* R_3 and R_3 *Overlaps* R_4. This approach first joins R_1 and R_2 for predicate *overlap*, the result with R_3 and the subsequent result with R_4 [1].
- ***All-Replicate*:** *All-Replicate* handles a join query by replicating all relations with replication function f_1 as defined in Section 4. A rectangle u is communicated to all cells which are in the fourth quadrant wrt. cell c_u. Each reducer then computes a part of the join output on the rectangles it receives.

  It should be noted that the decision of replicating the rectangles to cells in fourth quadrant is arbitrary. One can equivalently replicate the rectangles to first, second or third quadrant.

  Consider an output tuple $\mathcal{U}$. Naturally there may be multiple cells which will receive all the rectangles in $\mathcal{U}$ and all such cells can compute the output tuple $\mathcal{U}$. For example, consider the query $\mathcal{Q}_1$ as mentioned above and the rectangles as shown in Figure 3. Assume that there are 32 reducers and hence the 2D space is divided in a 8x4 grid. Consider the rectangle-set $\mathcal{U}$=(u_1,v_1,w_1,x_1). Rectangles in $\mathcal{U}$ satisfy the overlap conditions in $\mathcal{Q}_1$. As a result of replication with function f_1, reducers 19-24 and 27-32 will receive all rectangles in $\mathcal{U}$. No other reducer receives all four rectangles in $\mathcal{U}$ e.g., reducers 11-16 receive rectangles v_1, w_1 and x_1 but do not receive rectangle u_1. Potentially hence any reducer among 19-24, 27-32 can compute the output tuple $\mathcal{U}$. We need to hence employ a duplicate avoidance strategy so that an output tuple is computed exactly at one reducer. We next outline the duplicate avoidance strategy used.

6.2 Duplicate Avoidance Strategy

Consider an output tuple $\mathcal{U}$, let u_r, $u_r \in \mathcal{U}$ be the rightmost rectangle in $\mathcal{U}$ i.e., the rectangle with the largest x-coordinate of the starting-point and let u_l, $u_l \in \mathcal{U}$ be the lowermost rectangle in $\mathcal{U}$ i.e., the rectangle with the smallest y-coordinate of the starting point. Duplicates are avoided by letting only the partition-cell which contains the point ($u_r.x$, $u_l.y$) compute the output tuple $\mathcal{U}$.

Consider again the query $\mathcal{Q}_1$, Figure 3 and the rectangle-set $\mathcal{U}$=(u_1,v_1,w_1,x_1). x_1 is the rightmost rectangle in $\mathcal{U}$ and u_1 is the lowermost rectangle. Cell 19 contains the point ($x_1.x$, $u_1.y$) (as shown in Figure 3) and hence reducer 19 computes the output tuple $\mathcal{U}$. Reducers 20-24, 27-32 will determine that their respective cell do not contain the point ($x_1.x$, $u_1.y$) and hence will not output the tuple $\mathcal{U}$.

6.3 Why Splitting all Relations does not work?

Note that unlike a 2-way *overlap* join, a multi-way overlap join query can not be computed by splitting all rectangles. For example, again consider query $\mathcal{Q}_1$, Figure 3 and the rectangle-set $\mathcal{U}$=(u_1,v_1,w_1,x_1). Splitting each rectangle will imply that rectangle u_1 is received by reducer 18, rectangle v_1 is received by reducers 10 and 18, rectangle w_1 is received

[1]Assuming that this is the optimal order in which to evaluate 2-way joins

by reducers 2,3,10 and 11; and rectangle x_1 is received by reducers 3 and 11. Hence no one reducer receives all the rectangles and the output tuple $\mathcal{U}$ can not be computed.

Splitting both relations in a 2-way *overlap* join works as it is certain that two overlapping rectangles will be received at at-least one reducer. In a multi-way *overlap* join, rectangles forming an output tuple may be far apart and it is not certain that all rectangles in an output tuple will be received by at least one reducer. To circumvent this problem, *All-Replicate* replicates all rectangles to 4^{th} quadrant reducers which ensures that for each output tuple, there will be at least one reducer which will receive all rectangles for the output tuple. A suitable strategy can be adopted to avoid any duplicates (Section 6.2).

6.4 Why 2-way Cascade and All-Replicate are Naive?

Both these approaches are very inefficient. *2-way Cascade* handles a multi-way join as a cascade of 2-way joins. This hence produces a series of big intermediate join results and these big intermediate results are joined with subsequent relations. *2-way Cascade* hence executes a number of map and reduce tasks. The first problem with this approach is that it involves a huge reading and writing cost. For each successive 2-way join, a larger and larger size of data will be read (and written to the disk). Secondly a larger amount of data read naturally results in a larger communication cost among the cluster nodes.

All-Replicate carries out the multi-way join in one step and hence unlike *2-way Cascade* does not involve huge reading and writing cost. However *All-Replicate* involves a huge communication cost as it replicates each rectangle and as a result, each rectangle u is communicated to all the reducers which are in the 4^{th} quadrant wrt. rectangle u. Replicating each rectangle is a naive way of ensuring that there is atleast one reducer which will receive all the rectangles in an output tuple and can hence compute the output tuple.

However this naive way turns out to contain a lot of redundancy. Rectangles are replicated even if they do not form part of any output tuple resulting in unnecessary communication and processing e.g., a rectangle may not overlap with any other rectangle but is still replicated to multiple cells. For example, in Figure 3, rectangle u_4 will be communicated to all 32 reducers even though it does not form a part of any output tuple. Secondly for a majority of the output tuples, the overlapping rectangles are likely to be near-by and there is no provision of identifying that such output tuples can be computed locally e.g., rectangles (u_3,v_3,w_3,x_3) in Figure 3. We hence need an approach which carries out the multi-way join simultaneously (unlike a cascade of 2-way joins) but selectively replicates rectangles by taking into account their relative spatial locations. We next present *Controlled-Replicate*, a framework which achieves precisely this.

7. CONTROLLED-REPLICATE METHOD AND MULTI-WAY OVERLAP JOINS

In this section we present *Controlled-Replicate* framework and illustrate how we use it to compute multi-way overlap joins. We then present an experimental evaluation to show the efficacy of the *Controlled-Replicate* framework.

7.1 Motivation and Blueprint

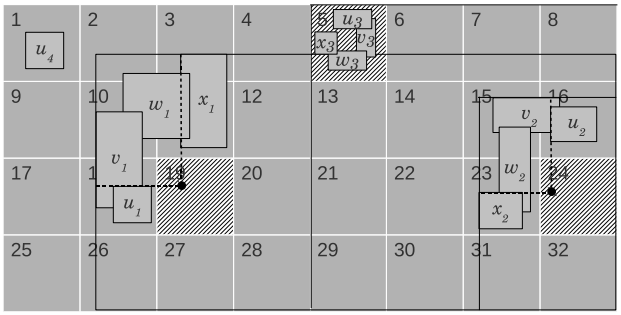


Figure 3: Example illustrating the scheme for deciding which reducer to compute the join

The basic idea of *Controlled-Replicate* framework is to identify which rectangles need not be replicated. The framework defines a number of conditions which each rectangle must satisfy. Any rectangle which does not satisfy these conditions is not replicated.

Controlled-Replicate runs as a round of two map-reduce jobs. First set of map-reduce operations figure out which rectangles need to be replicated. Map operations in the first round split all the relations and hence a reducer c_i receives all rectangles which overlap with cell c_i. Reduce operations in the first round then determine which rectangles need to be replicated. First round of reduce operations hence check which rectangles satisfy the conditions of *Controlled-Replicate* framework. A rectangle which is chosen for replication is marked with a flag. All other rectangles are output unmarked.

Second round of map-reduce operations carry out the replication and compute the multi-way join. Second round of map operations process the output of first round of reducers. The rectangles chosen for replication (i.e., which are marked) are replicated with function f_1 (see Section 4). Rest of the (unmarked) rectangles are projected. The second round of reducers compute the join. The duplicates are avoided using the strategy outlined in Section 6.2.

As only rectangles marked in the first round are replicated, *Controlled-Replicate* replicates much lesser number of rectangles as compared to *All-Replicate*. *Controlled-Replicate* hence incurs a much smaller communication cost as compared to *All-Replicate*. As *Controlled-Replicate* is carried out as a sequence of two map-reduce cycles, it incurs a much smaller reading and writing cost as compared to *2-way Cascade* which incurs a huge reading/writing cost due to generation of large intermediate results. We next outline the notion of consistency of a set of rectangles which is used in the conditions of *Controlled-Replicate* framework.

Notation: For simplicity purposes we refer to *Controlled-Replicate* and *All-Replicate* as *C-Rep* and *All-Rep* respectively.

7.2 Rectangle-Set

In this paper we will be heavily using the notion of a "rectangle set" or a "set of rectangles". We assume the following notations on such a set $\mathcal{U}$ used in this paper:

1. All rectangles in a set $\mathcal{U}$ belong to different relations.
2. We assume an order among the set $\mathcal{U}$ so that we can access i^{th} element of the set by the notion $\mathcal{U}$[i].

3. We say that the set $\mathcal{R}_s$ is the relation-set of rectangle-set $\mathcal{U}$ if the i^{th} rectangle in $\mathcal{U}$ belongs to i^{th} relation in $\mathcal{R}_s$ i.e., $\mathcal{U}$[i] $\in \mathcal{R}_s$[i] for all i. All the relations in $\mathcal{R}_s$ are distinct as no two rectangles in $\mathcal{U}$ come from the same relation (Point 1).

7.3 Consistency of a set of rectangles

Given a query $\mathcal{Q}$, we say that a set of rectangles $\mathcal{U}$ is consistent with its relation-set $\mathcal{R}_s$ if the following holds:

For any two indices j and k, if there is a condition (P_i, $\mathcal{R}_s[j]$, $\mathcal{R}_s[k]$) in query $\mathcal{Q}$ then the rectangles $\mathcal{U}$[j] and $\mathcal{U}$[k] satisfy the predicate P_i.

The notion of consistency of a set of rectangles implies that the rectangles from set $\mathcal{U}$ satisfy all the conditions in query $\mathcal{Q}$ which are formed by the relations in relation-set $\mathcal{R}_s$.

Consider the rectangles shown in Figure 3 and the query $\mathcal{Q}_1$: R_1 *Overlaps* R_2 and R_2 *Overlaps* R_3 and R_3 *Overlaps* R_4. Let the rectangles in relations R_1, R_2, R_3 and R_4 be represented using u, v, w and x respectively. The rectangle-set $\mathcal{U}_1$=(u_1,v_1,w_1) is consistent with the relation-set $\mathcal{R}_s$=(R_1, R_2, R_3) as rectangle u_1 and v_1 overlap and rectangles v_1 and w_1 overlap. This is required by the presence of join conditions 'R_1 *Overlaps* R_2' and 'R_2 *Overlaps* R_3' in query $\mathcal{Q}$. There is no condition R_1 *Overlaps* R_3 in $\mathcal{Q}_1$ and hence the overlap of rectangles u_1 and w_1 is not required for consistency. The rectangle-set $\mathcal{U}_1$=(u_2,v_1,w_1) is not consistent with $\mathcal{R}_s$=(R_1,R_2,R_3) as rectangles u_2 and v_1 do not overlap but this is required by the presence of condition 'R_1 *Overlaps* R_2' in query $\mathcal{Q}$.

The intuition behind the notion of consistency is that each subset of an output tuple must be consistent. For example, consider the output tuple $\mathcal{U}$=(u_1,v_1,w_1,x_1) and query $\mathcal{Q}_1$. Each subset of the tuple $\mathcal{U}$ is consistent. One of the condition of *C-Rep* is that the rectangles which are replicated must be part of some consistent rectangle-sets.

7.4 Conditions of Controlled-Replicate

We now describe the conditions of *Controlled-Replicate* framework which are used to find rectangles to be replicated. Let the query be $\mathcal{Q}$ and its relation-set $\mathcal{R}$. Let $\mathcal{U}_c$ be the set of rectangles split on partition-cell c and subsequently received by the reducer c. The reducer c first identifies all rectangle-sets $\mathcal{U}$ which satisfy the following conditions. Let $\mathcal{R}_s$ be the set of relations to which the rectangles in $\mathcal{U}$ belong to.

1. **C1:** $\mathcal{U}$ is consistent with $\mathcal{R}_s$.
2. **C2:** Consider two relations R_1 and R_2, $R_1 \in \mathcal{R}_s$, $R_2 \notin \mathcal{R}_s$ and $R_2 \in \mathcal{R}$ s.t. there is a triple (Ov,R_1,R_2) or (Ov,R_2,R_1) in $\mathcal{Q}$. Let u_1 be the rectangle in $\mathcal{U}$ which belongs to relation R_1. For all such pairs R_1 and R_2, the rectangle u_1 crosses the boundary of partition-cell c i.e., rectangle u_1 overlaps with a partition-cell other than c.
3. **C3:** There must be at least one pair (R_1,R_2) satisfying the conditions C2.
4. **C4:** No superset of $\mathcal{U}$ satisfies conditions C1, C2 and C3.

Let $\mathcal{US}_c$ be the set of such rectangle-sets. We define uS_c as the union of all rectangle sets in $\mathcal{US}_c$ i.e.,

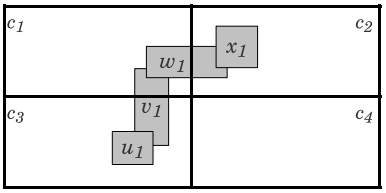


Figure 4:

$$uS_c = \bigcup_{\mathcal{U}} \mathcal{U},\ s.t.\ \mathcal{U} \in \mathcal{US}_c$$

C-Rep replicates all rectangles in uS_c which start in partition-cell c i.e., the start-point of rectangle u lies in cell c.

7.5 Proof-of-Correctness

We next show that this approach will compute the join output correctly. Consider the rectangles in an output tuple $\mathcal{U}'$=(u_1,u_2,...,u_m), $u_i \in R_i$. Consider the set of rectangles $\mathcal{U}_c^{'}$, $\mathcal{U}_c^{'} \subseteq \mathcal{U}'$ which are split or projected onto the partition-cell c. Let $\mathcal{R}_c$ be the set of relations to which the rectangles in $\mathcal{U}_c^{'}$ belong. As an output tuple is by default consistent with $\mathcal{R}$, the set of rectangles $\mathcal{U}_c^{'}$ will be consistent with $\mathcal{R}_c$. This satisfies the condition C1. So we next need to show that $\mathcal{U}_c^{'}$ satisfies the conditions C2 and C3.

Consider a join condition represented by a triple (P,R_j,R_k) or (P,R_k,R_j) in $\mathcal{Q}$ s.t. $R_j \in \mathcal{R}_c$, $R_k \notin \mathcal{R}_c$ and $R_k \in \mathcal{R}$. Let u_k and u_j be the corresponding rectangles in $\mathcal{U}'$, i.e., $u_k \in R_k$ and $u_j \in R_j$. As u_j and u_k are present in the output tuple, they must overlap. However the rectangle u_k does not belong to $\mathcal{U}_{c'}$ as $R_k \notin \mathcal{R}_c$. Rectangle u_j and u_k hence can overlap only if rectangle u_j crosses-over the boundary of cell c. Condition C2 represents this case.

However if no such join-condition exists in query $\mathcal{Q}$, it implies that all rectangles in $\mathcal{U}'$ overlap with partition-cell c. The cell c hence can itself compute the output tuple $\mathcal{U}'$ and does not need to replicate any rectangle. Condition C3 represents this boundary case.

Condition C4 is hence present only for efficiency purposes. If a rectangle-set $\mathcal{U}$ satisfies conditions C1-C3, all of its subsets will also satisfy conditions C1-C3.

7.6 Intuition behind Controlled-Replicate

Condition C1 conveys that a rectangle-set $\mathcal{U}$ must be consistent for inclusion in $\mathcal{US}_c$ (Section 7.3). It is because each subset of an output tuple is by default consistent. If a rectangle-set $\mathcal{U}$ is not consistent, it can not be part of an output tuple.

Condition C2 conveys that if the number of rectangles in a consistent rectangle-set $\mathcal{U}$ is less than the number of relations involved in query $\mathcal{Q}$ (i.e., the cardinality of $\mathcal{R}$), then certain rectangles in $\mathcal{U}$ must cross-over the partition-cell boundary. Consider the join condition in $\mathcal{Q}$ - (P, R_1, R_2) s.t. $R_1 \in \mathcal{R}_s$ and $R_2 \notin \mathcal{R}_s$ and let u be the rectangle in $\mathcal{U}$ belonging to relation R_1. If rectangle u does not cross the partition-cell c, then u can not overlap with any rectangle v which does not overlap with cell c. $\mathcal{U}$ hence must not be included in $\mathcal{US}_c$ even though it is consistent. By not including $\mathcal{U}$ in $\mathcal{US}_c$, *C-Rep* will possibly avoid replicating some rectangles [2].

[2]Possibly, because a rectangle u in $\mathcal{U}$ may still get replicated as it may be a part of another rectangle-set in $\mathcal{US}_c$

Condition C3 conveys that the reducer c may receive all rectangles to form an output tuple. Such a rectangle-set is consistent and also satisfies condition C2 as no rectangle needs to cross-over. However this must not be included in $\mathcal{US}_c$ as reducer c will receive these rectangles in second round and can compute this output tuple. Condition C3 hence eliminates this boundary case.

Consider the simple scenario of Figure 4 and query $\mathcal{Q}_1$ (Section 7.3). Let the rectangles of relations R_1, R_2, R_3, R_4 be represented by u,v,w and x respectively. (u_1, v_1, w_1, x_1) form an output tuple. Reducer c_1 receives rectangles v_1 and w_1 and needs to decide whether to replicate these rectangles or not. The set (v_1, w_1) is consistent with its relation-set (R_2, R_3) as v_1 and w_1 overlap. If v_1 and w_1 were not overlapping, reducer C_1 would have deduced that v_1 and w_1 together can not be part of an output tuple (Condition C1).

Secondly both rectangles v_1 and w_1 cross over the cell c_1. They can hence overlap with other rectangles not overlapping with cell c_1 (u_1 and x_1 in this case) and hence can form an output tuple (Condition C2). If any of the v_1 or w_1 were not crossing over, reducer c_1 would have deduced that v_1 and w_1 together can not form an output tuple with rectangles not overlapping with cell c_1. Reducer c_1 hence finds that the rectangle-set (v_1, w_1) is a candidate-set that can belong to an output tuple and replicates both the rectangles v_1 and w_1. Note that the output tuple (u_1, v_1, w_1, x_1) will be computed by reducer c_4.

7.7 Illustrative Example

We next illustrate the working of *C-Rep* approach using a detailed example. Consider the rectangles shown in Figure 5 with the whole 2D space partitioned into 4 cells and the query $\mathcal{Q}_1$ (Section 7.3). Rectangles belonging to relations R_1, R_2, R_3 and R_4 are represented using u, v, w and x respectively. For the rectangles shown in Figure 5 the output will consist of tuples (u_2, v_3, w_1, x_1), (u_2, v_3, w_1, x_2), (u_3, v_3, w_1, x_1) and (u_3, v_3, w_1, x_2) . Consider the reducer c_1. After splitting every rectangle, the reducer c_1 receives the rectangles u_1, v_1, v_2, v_3, x_2, u_2 , w_1 and v_4 in the first stage. The set $\mathcal{US}_{c_1}$ is [(u_2, v_3, w_1), (v_3, w_1, x_2), (v_4)] as these three sets satisfy all four conditions of *C-Rep*. The set uS_{c_1} is hence (u_2, v_3, v_4, w_1, x_2) and reducer c_1 replicates all these rectangles as they all start-out within c_1.

The rectangle-set $\mathcal{U}_1$=(u_2, v_3, w_1) is consistent with relation set $\mathcal{R}_{s1}$=(R_1, R_2, R_3) as rectangles in $\mathcal{U}_1$ satisfy the overlap predicates formed by relations in $\mathcal{R}_{s1}$. Rectangles u_2 and v_3 overlap and rectangles v_3 and w_1 overlap. As there is the join-condition R_3 *overlaps* R_4 in query $\mathcal{Q}$, the condition C2 of *C-Rep* requires that the rectangle w_1 crosses the partition-cell boundary and it does. The rectangle-set $\mathcal{U}_1$ is hence included in set $\mathcal{US}_{c_1}$. The inclusion of the set $\mathcal{U}_1$ in $\mathcal{R}_{s1}$ signifies that there might be an output tuple of form (u_2, v_3, w_1, x), $x \in R_4$ s.t. x does not intersect with cell c_1. In the Figure 5, (u_2, v_3, w_1, x_1) is such an output tuple. Supposing x_1 were not present the reducer would have still needed to replicate the rectangles in $\mathcal{U}_1$ as the reducer c_1 does not have any way to know this apriori.

Similar is the case for rectangle-set $\mathcal{U}_2$=(v_3, w_1, x_2). $\mathcal{U}_2$ is consistent with relation-set $\mathcal{R}_{s2}$=(R_2, R_3, R_4). The presence of join-condition R_1 *overlaps* R_2 requires that the rectangle v_3 crosses the cell-boundary and it does.

The rectangle-set $\mathcal{U}_3$=(v_4) is consistent with relation-set (R_2) and it crosses the cell-boundary of C_1 as required by conditions R_1 *overlaps* R_2 and R_2 *overlaps* R_3.

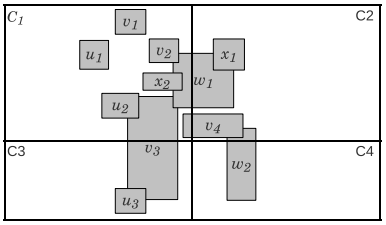


Figure 5: *Controlled-Replicate* Example

Consider the rectangle-set $\mathcal{U}_4$=(u_2, v_3, w_1, x_2). This set belongs to the output and can be computed by the reducer c_1 itself. $\mathcal{U}_4$ hence violates the condition C3 of *C-Rep*. Next consider the set $\mathcal{U}_5$=(v_2, w_1). This is consistent with relation-set (R_2, R_3) however the rectangle v_2 does not cross the cell-boundary as required due to the join condition R_1 *overlaps* R_2. $\mathcal{U}_5$ hence violates the condition C2 of *C-Rep*.No other rectangle-sets satisfy all conditions of *C-Rep*.

For reducer c_3 the rectangle set $\mathcal{U}_6$=(u_3,v_3) satisfies all conditions of *C-Rep*. However the reducer c_3 replicates only the rectangle u_3 as the rectangle v_3 does not start within c_3.

The output tuples (u_2, v_3, w_1, x_1), (u_2, v_3, w_1, x_2), (u_3, v_3, w_1, x_1) and (u_3, v_3, w_1, x_2) are computed by reducers c_2, c_1, c_4 and c_3 respectively in the second stage.

Table 3: Query $\mathcal{Q}_2$, Varying rectangle dimensions

nI=2 million, dS=Uniform, dX, dY, dL, dB=Uniform (x_{min},x_{max})=(0,100K), (y_{min}, y_{max})=(0,100K)					
l_{max}, b_{max}	Time *2-way Cascade* (hh:mm)	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	# Recs Rep. *C-Rep* (million)	# Recs Rep. *C-Rep-L* (million)
100	00:10	00:07	00:07	0.11, (7.6)	0.11 (6.1)
200	00:13	00:09	00:08	0.25, (10.1)	0.25 (6.5)
300	00:30	00:16	00:13	0.39, (12.0)	0.39 (6.8)
400	02:23	00:28	00:20	0.53, (14.5)	0.53 (7.1)
500	05:14	00:59	00:33	0.67 (16.8)	0.67 (7.3)

7.8 Experimental Evaluation

In this section, we show the efficacy of *C-Rep* approach over naive approaches first on synthetic and then on a real-life data-set. We first describe the cluster set-up, how we generate the synthetic data and the details of real-life California Road data.

7.8.1 Experimental SetUp

The experiments are run over a 16 core Hadoop cluster built using Blade Servers with three 3 GHz Xeon processors having 8GB memory and 200 GB SATA drives. These machines run Red Hat Linux 5.2. The software stack comprises of Hadoop 0.20.2 with HDFS. All the experiments are executed with 64 reduce processes. The 2D space is hence divided in 8x8 grid.

7.8.2 Datasets

Generation of Synthetic Data: We use synthetic data for experimental evaluation. We write a script to generate a set of rectangles. The parameters to this script are the following: (a) Number of rectangles (nI), (b) Distribution of x and y coordinates of start-point of rectangles (dX and dY). (c) Distribution of rectangle length and breadth (dL and dB), (d) x and y ranges within which all rectangles lie ($[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$) (e) Minimum and maximum length and breadth of rectangles ($[l_{min}, l_{max}]$, $[b_{min}, b_{max}]$).

Table 2: Query $\mathcal{Q}_2$, Varying the dataset size

dS=Uniform, dX, dY, dL, dB=Uniform, (x_{min},x_{max})=(0,100K) (y_{min}, y_{max})=(0,100K), (l_{min},l_{max})=(0,100), (b_{min},b_{max})=(0,100)							
nI # Tuples (million)	Time *2-way Cascade* (hh:mm)	Time *All-Rep* (hh:mm)	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	# Rectangles Rep. *All-Rep* (millions)	# Rectangles Rep. *C-Rep* (millions)	# Rectangles Rep. *C-Rep-L* (millions)
1	00:05	00:32	00:05	00:05	3, (64.3)	0.05, (3.9)	0.05 (3.0)
2	00:10	01:22	00:07	00:07	6, (128.7)	0.1, (7.6)	0.1 (6.1)
3	00:13	>03:00	00:08	00:09	9, (-)	0.19, (12.5)	0.19 (9.2)
4	00:24	>03:00	00:11	00:11	12, (-)	0.23, (15.6)	0.23 (12.2)
5	00:35	>03:00	00:15	00:13	15, (-)	0.31 (19.8)	0.31 (17.9)

The computational effort needed to compute the join can be effected by varying these parameters e.g., increasing number of rectangles results in a larger output size. If l_{max} or b_{max} increases, more rectangles will overlap and the output size will be larger. We can hence carry out a number of controlled-experiments to investigate the efficacy of approaches presented in this paper.

Details of real-life California Road Data: Census 2000 TIGER/Line shape files were used to create the real data set [1]. Road data layer information in the California state was used to generate MBBs in the real data set. The total number of road objects in the California dataset is 2092079. For each road object in the shape file, given bounds were used to create MBBs. We then converted the latitude and longitude to a flat XY co-ordinate system using the Openmap library [2] and the ratio |x-range|/|y-range| was found to be 0.63. The flattened MBBs were used for all the join computations in our experiments.

The mapping was done on 2D space with y-range as [0, 100K] and x-range as [0,63K]. Average MBB length and breadth were found to be 18 and 8 respectively. Minimum and maximum MBB length were found to be 1 and 2285 respectively while minimum and maximum MBB breadth were found to be 1 and 1344 respectively. Minimum and maximum MBB areas were found to be 1 and 3071K respectively. 97% of rectangles have both length and breadth less than 100 while 99% of the rectangles have both their length and breadth less than 1000.

7.8.3 Metrics Computed

In each experiment we report the following metrics:

1. **Time Taken:** This is the end-to-end time for running an algorithm. It covers cost of reading and parsing data, communication cost among the cluster nodes, join processing cost as well as the cost of writing intermediate outputs on HDFS.

2. **The number of Rectangles Replicated:** This is the number of rectangles marked by *C-Rep* for replication.

3. **The number of Rectangles After Replication:** A rectangle is communicated to multiple reducers after replication. This is the aggregated count of rectangles communicated to reducers after replication. For example, in Figure 5, the rectangle w_1 after replication, is communicated to reducers C_1, C_2, C_3 and C_4. The rectangle w_1 hence is counted once in the metric 'the number of rectangles replicated' and four times in the metric 'the number of rectangles communicated after replication'. This metric is hence much larger than the number of rectangles replicated in *C-Rep*. In all experiments this metric is mentioned within parenthesis.

7.8.4 Varying DataSet Size - Synthetic Data

We consider the query $\mathcal{Q}_2 = R_1$ *overlaps* R_2 *and* R_2 *overlaps* R_3. Table 2 presents a comparison of all three approaches, *2-way Cascade*, *All-Rep* and *C-Rep* (Ignore *C-Rep-L* at the moment). We generate three sets of rectangles synthetically, R_1, R_2 and R_3. Parameter values using which the synthetic data is generated are also provided in Table 2. All three relations contain same number of tuples. The number of tuples in all three relations are varied from 1 million to 5 million. Both *2-way Cascade* and *All-Rep* are found to be taking much longer time vis-a-vis *C-Rep*. The time required for both of these increases rapidly as the size of the relations increase. *2-way Cascade* solves a multi-way join query as a series of 2-way joins which involves joining large intermediate results and hence requires a much higher time to complete.

All-Replicate replicates all rectangles and hence incurs a much higher communication cost. In comparison, *C-Rep* replicates a much smaller number of rectangles. Table 2 also presents the number of rectangles replicated for both the approaches. The numbers in parentheses represent the number of rectangles after the replication (Section 7.8.3). For nI=1 million, *All-Rep* communicates an aggregated 64.3m rectangles while *C-Rep* communicates only 3.9m rectangles. It is this huge reduction in the number of communicated rectangles which shows up in much improved performance of *C-Rep* over *All-Rep*. Both the communication cost as well as the time required for computing the partial join output at each reducer are smaller for *C-Rep* vis-a-vis *All-Rep*. We do not present the numbers for *All-Rep* in the remaining experiments as it is clear that *C-Rep* will significantly outperform *All-Rep* .

7.8.5 Varying Rectangle Size - Synthetic Data

We next keep the number of rectangles in each dataset at 2m while varying l_{max} and b_{max}. As rectangles of larger and larger dimensions are allowed, more number of rectangles overlap and larger is the size of output. The results are presented in Table 3. The *C-Rep* approach easily outperforms the naive *2-way Cascade*.

7.8.6 Experiments on California Road Data

We first define the notion of enlarging a rectangle by factor k.

Enlarging a rectangle by factor k: Let (x_1, y_1) be the top-left vertex of a rectangle and let (x_2, y_2) be the bottom-right vertex . The top-left vertex of the enlarged rectangle by factor k is given by $(x_1 - (x_2 - x_1) * (k-1)/2, y_1 + (y_2 - y_1) * (k-1)/2)$ and the bottom-right vertex is given by $(x_2 + (x_2 - x_1) * (k-1)/2, y_2 - (y_2 - y_1) * (k-1)/2))$. In other words, the length and breadth of each rectangle is increased by a factor of k keeping the center of the rectangle

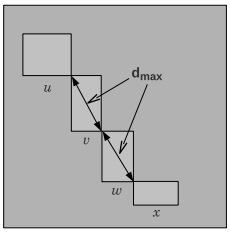


Figure 6: Controlled Replication In Limit

identical.

In this experiment we compare the performance of *2-way Cascade* and *C-Rep* on real-life California road data. We consider the star join query $\mathcal{Q}_{2s}$=*R Overlaps R* and *R Overlaps R*. This hence find road triples (rd_1,rd_2,rd_3) such that rd_1 overlaps with rd_2 and rd_2 overlaps with rd_3. We take California Road data and construct different datasets from this by enlarging each rectangle by a factor k. Hence as the enlarging factor increases, the sizes of the MBBs increase and more and more MBBs will overlap resulting in a larger output. Table 4 provides the comparison. We again find that *C-Rep* easily outperforms *2-way Cascade*.

Table 4: Query $\mathcal{Q}_{2s}$, California Road Data

nI=2 million (x_{min},x_{max})=(0,63K), (y_{min}, y_{max})=(0,100K)					
Enlarging Factor	Time *2-way Cd* (hh:mm)	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	#Recs Rep. *C-Rep* (million)	#Recs Rep. *C-Rep-L* (million)
1.00	00:19	00:15	00:14	0.08, (0.8)	0.08 (0.64)
1.25	00:27	00:24	00:21	0.12, (0.9)	0.12 (0.65)
1.5	00:43	00:25	00:24	0.18, (1.0)	0.18 (0.66)
1.75	01:04	00:46	00:42	0.23, (1.14)	0.23 (0.67)
2.0	01:35	00:57	00:53	0.32 (1.33)	0.32 (0.68)

7.9 Controlled Replicate in Limit

C-Rep approach replicates a rectangle u starting out in partition-cell c to all reducers which lie in the 4^{th} quadrant wrt partition-cell c. However a rectangle is only likely to join with rectangles which start-out in neighboring partition-cells. Rectangle u hence is not likely to be present in the join outputs of many reducers. Performance of *C-Rep* hence can be improved if we can apriori decide which reducers won't require a rectangle and avoid replicating the rectangle to those reducers. We call this approach as *Controlled Replicate in Limit (C-Rep-L)*.

We can design one such scheme if we know the upper bound on the diagonal length of the rectangles for each relation. Lets say d_{max} is the upper bound on the diagonal-length for each relation. If number of relations in the query are m, a rectangle u being replicated by *C-Rep* need only be communicated to reducers which are in the 4^{th} quadrant to rectangle u and are at most a distance of $(m-2) * d_{max}$ from the rectangle u [3].

Consider the figure 6 and query $\mathcal{Q}_1$ (Section 7.3). Suppose d_{max} is an upper-bound on the diagonal-length of any rectangle. Then it suffices that rectangles of relation R_1 and R_4 are replicated within distance $2d_{max}$ while rectangles of relations R_2 and R_3 are replicated within distance d_{max} i.e., the replication function f_2 is applied with distance parameter $2d_{max}$ for relations R_1 and R_4 while with parameter d_{max} for R_2 and R_3. Consider the output tuple (u,v,w,x). Rectangles u and x can at most be distance $2d_{max}$ apart as the maximum diagonal length is d_{max} (Figure 6). Hence rectangle u (and x) needs to be replicated to all cells in 4^{th} quadrant which are at most $2d_{max}$ distance apart from rectangle u. Similarly rectangles v and x (u and w) can be at most d_{max} apart.

C-Rep hence limits the number of rectangles being replicated and *C-Rep-L* limits the extent of replication of rectangles chosen by *C-Rep* for replication.

[3]It is assuming that the query is a chain join. The bounds can be easily obtained for a general query graph. A graph formalism is avoided in this paper so as to avoid mathematical details not directly relevant to this paper.

7.10 Experimental Evaluation-2

We implement *C-Rep-L* as described above and repeat the two controlled-experiments on synthetic data outlined in Section 7.8.4 and 7.8.5. Tables 2 and 3 also list out the results for *C-Rep-L*. The first experiment varies the number of rectangles. Not much difference is observed between *C-Rep* and *C-Rep-L* as the number of rectangles after replication (Section 7.8.3) i.e., the numbers in parenthesis are of similar magnitude. Note that the number of replicated rectangles remain the same. *C-Rep-L* only determines the limit to which a rectangle is replicated to. The decision whether a rectangle needs to be replicated or not is made by *C-Rep*.

However a large improvement is observed in the performance of *C-Rep-L* vis-a-vis *C-Rep* for the case when we vary the maximum rectangle dimension (Table 3). As the l_{max} and b_{max} increase, the number of overlapping rectangles also increase. There is a substantial difference between the number of rectangles after replication (numbers in parenthesis) between *C-Rep* and *C-Rep-L*. *C-Rep-L* involves much smaller number of rectangles after replication and this translates to a smaller processing time. This shows the efficacy of *C-Rep-L* in identifying and eliminating unnecessary replication. Overall *C-Rep-L* substantially improves over the naive approach of *2-way Cascade*.

Finally Table 4 presents the performance of *C-Rep-L* on California road data. A small improvement is observed in the performance of *C-Rep-L* over *C-Rep*. The improvement is small as there is not much difference between the number of rectangles after replication (number in parenthesis).

8. HANDLING RANGE JOINS

In this section we present *Controlled-Replicate* based approaches for handling multi-way range join queries. The selectivity of *range* predicate is much lower as compared to *overlap* predicate as a *range* predicate joins rectangles at some distance apart while *overlap* predicate joins rectangles which are overlapping.

The conditions for *C-Rep* for *range* join queries remain the same except condition C2 outlined below. We are looking for properties a rectangle-set $\mathcal{U}$ must satisfy in cell c.

Condition C2 in *Controlled-Replicate* for Range Joins: Consider two relations R_1 and R_2, $R_1 \in \mathcal{R}_s$, $R_2 \notin \mathcal{R}_s$ and $R_2 \in \mathcal{R}$ s.t. there is a triple ($Ra(d)$,R_1,R_2) or ($Ra(d)$,R_2,R_1) in $\mathcal{Q}$. Let u_1 be the rectangle in $\mathcal{U}$ which belongs to relation R_1. For all such pairs R_1 and R_2, there must be a cell c' ($\neq$ c) which is within distance d from rectangle u_1 i.e. $dist(c', u_1) \leq d$.

The proof-of-correctness is along the similar lines as outlined in Section 7.4. The revision in the condition C2 is natural as if a rectangle u_1 starting within cell c is within range d of another rectangle u_2 starting in cell c' then cell c' must be within distance d from u_1. If not, rectangle u_1

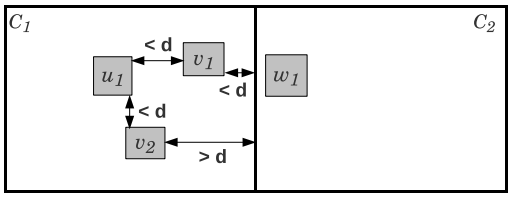


Figure 7: Controlled Replication for Range Queries

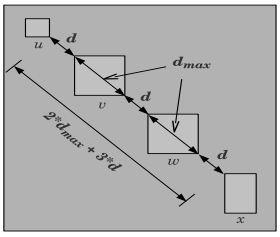


Figure 8: C-Rep In Limit for Range Queries

can have range relationship only with rectangles starting in cell c and need not be replicated.

For example, consider the query $\mathcal{Q}_3$=R_1 $Ra(d)$ R_2 and R_2 $Ra(d)$ R_3 and Figure 7. Let the rectangles denoted by u, v and w belong to relations R_1, R_2 and R_3. Reducer C_1 will receive rectangle u_1, v_1 and v_2 in the first round. Consider the reducer C_1 and rectangle-set (u_1, v_1). As the cell C_2 is less than distance d apart from rectangle v_1, rectangle v_1 can be within distance d of a rectangle ($\in R_3$) starting out in cell C_2. For example rectangle v_1 is within distance d of rectangle w_1. Rectangle-set (u_1, v_1) is consistent as rectangle u_1 is within distance d of rectangle v_1 and hence reducer C_1 will mark rectangles u_1 and v_1 for replication. Note that even if the rectangle w_1 were more than distance d apart from rectangle v_1, u_1 and v_1 would have still required to be replicated as reducer C_1 has no way to figure that out. Rectangle v_2 will not be replicated as no cell is within distance d of v_2 and hence condition C2 as described above is not satisfied.

Controlled-Replicate-In-Limit: We next extend the arguments in Section 7.9 in case of range joins. We need to find the distance to which a rectangle marked by *C-Rep* needs to be replicated to. If the number of relations in the query are m, d_{max} is the upper-bound on the diagonal length of rectangles and d is the upper-bound on all the range parameters in the query then a rectangle u needs to be replicated to all cells in the 4^{th} quadrant which are within a distance $(m-2)$*d_{max}+$(m-1)$*d from rectangle u.

Consider Figure 8 and query R_1 $Ra(d)$ R_2 and R_2 $Ra(d)$ R_3 and R_3 $Ra(d)$ R_4. Let rectangles u, v, w and x belong to relations R_1, R_2, R_3 and R_4 respectively. Rectangles u and x can be part of an output tuple only if they are within distance 2*d_{max}+3*d as shown in Figure 8. Hence a rectangle belonging to relations R_1 and R_4 need to be replicated to 4^{th} quadrant cells within distance 2*d_{max}+3*d.

Similarly rectangles belonging to relations R_2 and R_3 need to be replicated to 4^{th} quadrant cells within distance d_{max} + 2*d. Rectangles v and x can be part of an output tuple only if they are within distance d_{max}+2*d. Rectangles v and w or rectangles v and u can be part of an output tuple only if they are within distance d. Hence R_2 rectangles must be replicated to distance d_{max}+2*d. Similar is the reasoning for rectangles belonging to relation R_3.

8.1 Experimental Evaluation

We now present an experimental evaluation on both synthetic as well as California Road data. We consider the query $\mathcal{Q}_3$=R_1 $Ra(d)$ R_2 and R_2 $Ra(d)$ R_3 with d=100. We first generate three sets of rectangles synthetically.

In the first experiment, we vary the number of rectangles in each relation while keeping other parameters identical. Table 5 presents the results. The parameters using which the synthetic data is generated are also provided. Performance numbers for *2-way Cascade* quickly spiral out. *C-Rep-L* provides much better performance vis-a-vis both *2-way Cascade* and *C-Rep*. It is again due to the fact that *C-Rep-L* carries out much lesser redundant replication. Numbers of rectangles after replication (Section 7.8.3) i.e. the numbers in parenthesis, are found to be approx. 30% of the corresponding numbers for *C-Rep*. There is hence a huge difference between the number of communicated rectangles between *C-Rep* and *C-Rep-L*. For example, for nI=5m, *C-Rep* communicates 40 million pairs more than those communicated by *C-Rep-L* and this difference exhibits as a much improved performance of *C-Rep-L*.

In the second experiment, we vary the distance parameter d and process query $\mathcal{Q}_3$. Number of rectangles are fixed at 1 million in all relations. Table 6 presents the results. *C-Rep-L* clearly out-performs *C-Rep*. It is again due to the fact that the number of rectangles after replication are much smaller for *C-Rep-L* as compared to *C-Rep*.

Finally, we present performance results on California Road Data in Table 7. We consider the query: $\mathcal{Q}_3 s$=R $Ra(d)$ R and R $Ra(d)$ R. This finds the road triples (rd_1,rd_2,rd_3) such that rd_1 is within distance d from rd_2 and rd_2 is within distance d from rd_3. We sample the road data with probability 0.5 and retain only 1 million road MBBs. We then vary the distance parameter d. *C-Rep* beats *2-way Cascade* easily. *C-Rep-L* performs slightly better than *C-Rep* as there is a slight reduction in the number of rectangles after replication.

Table 5: Query $\mathcal{Q}_3$, Varying the dataset size

dS=Uniform, dX, dY, dL, dB=Uniform, (x_{min}, x_{max})=(0,100K) (y_{min}, y_{max})=(0,100K), (l_{min}, l_{max}) & (b_{min}, b_{max})=(0,100) nI, # Rectangles Replicated in millions					
nI	Time *2-way Cd* (hh:mm)	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	# Rectangles Replicated *C-Rep*	# Rectangles Replicated *C-Rep-L*
1	00:11	00:10	00:06	0.36, (9.1)	0.36 (3.0)
2	00:56	00:27	00:12	0.61, (16.5)	0.61 (6.1)
3	02:27	01:12	00:23	0.96 , (26.2)	0.96 (9.7)
4	04:23	01:43	00:39	1.3 , (41.6)	1.3 (12.8)
5	>06:00	02:37	01:03	1.7 (58.4)	1.7 (15.8)

Table 6: Query $\mathcal{Q}_3$, Varying distance parameter d

dS, dX, dY, dL, dB=Uniform, (x_{min}, x_{max})=(0,100K) (y_{min}, y_{max})=(0, 100K), (l_{min},l_{max}) & (b_{min},b_{max})=(0,100) nI = 1 million, # Rectangles Replicated in millions				
d	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	# Rectangles Replicated *C-Rep*	# Rectangles Replicated *C-Rep-L*
100	00:10	00:06	0.36, (9.1)	0.36 (3.0)
200	00:18	00:08	0.53, (13.1)	0.53 (3.2)
300	00:42	00:15	0.72, (16.5)	0.72 (3.3)
400	01:16	00:25	0.94, (20.3)	0.94 (3.4)
500	01:40	00:41	1.06, (24.8)	1.06 (3.5)

9. MULTI-WAY HYBRID JOIN QUERIES

The *Controlled-Replicate* framework easily extends to handle join queries involving both *overlap* and *range* predicates. Condition C2 can be written down as follows. We are looking for properties, a rectangle-set $\mathcal{U}$ must satisfy in cell c.

Table 7: Query $\mathcal{Q}_3 s$, California Road Data

nI=1m, (x_{min}, x_{max})=(0,63K) (y_{min}, y_{max})=(0,100K) nI, # Rectangles Replicated in millions					
d	Time *2-way Cascade* (hh:mm)	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	# Recs Rep. *C-Rep*	# Recs Rep. *C-Rep-L*
5	01:16	00:14	00:11	0.04, (4.1)	0.36 (3.1)
10	02:02	00:21	00:16	0.07, (4.9)	0.61 (3.2)
15	02:52	00:36	00:23	0.09, (5.4)	0.96 (3.2)
20	04:06	00:46	00:31	0.10, (5.9)	1.3 (3.3)

Condition C2 in *Controlled-Replicate* for Hybrid Queries: Consider two relations R_1 and R_2, $R_1 \in \mathcal{R}_s$, $R_2 \notin \mathcal{R}_s$ and $R_2 \in \mathcal{R}$ s.t. there is a triple (P,R_1,R_2) or (P,R_2,R_1) in $\mathcal{Q}$. Let u_1 be the rectangle in $\mathcal{U}$ which belongs to relation R_1. For all such pairs R_1 and R_2, the following holds:

1. If predicate P is an *overlap* predicate, the rectangle u_1 crosses the boundary of partition-cell c.

2. If predicate P is a *range* predicate with distance parameter d, there must be a cell c' ($\neq$ c) which is within distance d from rectangle u_1 i.e. $dist(c', u_1) \leq d$.

Condition C2 is a natural union of the individual C2 conditions presented in Section 7 and Section 8. Alternatively a hybrid query can be handled by replacing the overlap predicate as a range predicate with distance parameter 0 and handling the resulting query as a multi-way range join query. Similarly for *C-Rep-L* we can derive the distance bounds for rectangles belonging to each relation.

Table 8: Query $\mathcal{Q}_4$, Varying the dataset size

dS=Uniform, dX, dY, dL, dB=Uniform, (x_{min}, x_{max})=(0,100K) (y_{min}, y_{max})=(0,100K), (l_{min}, l_{max}) & (b_{min}, b_{max})=(0,100) nI, # Rectangles Replicated in millions, d=200				
nI # Recs	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	# Rectangles Replicated *C-Rep*	# Rectangles Replicated *C-Rep-L*
1	00:07	00:06	0.27, (8.0)	0.27 (3.1)
2	00:16	00:12	0.57, (15.8)	0.57 (6.3)
3	00:39	00:23	0.94 , (26.5)	0.94 (9.6)
4	01:08	00:44	1.22 , (33.0)	1.22 (12.7)
5	01:57	01:16	1.54, (46.3)	1.54 (16.1)

Table 9: Query $\mathcal{Q}_4 s$, California Road Data

dS=Uniform, dX, dY, dL, dB=Uniform, (x_{min}, x_{max})=(0,63K) (y_{min}, y_{max})=(0,100K) nI=1 million, # Rectangles Replicated in millions				
d	Time *C-Rep* (hh:mm)	Time *C-Rep-L* (hh:mm)	# Rectangles Replicated *C-Rep*	# Rectangles Replicated *C-Rep-L*
10	00:28	00:26	0.08, (5.0)	0.08 (3.6)
20	00:39	00:30	0.11, (5.9)	0.11 (3.8)
30	00:51	00:41	0.14 , (6.7)	0.14 (3.9)
40	01:03	00:48	0.18 , (7.5)	0.18 (4.1)

9.1 Experimental Evaluation

We consider the query $\mathcal{Q}_4$=R_1 *Ov* R_2 and R_2 *Ra*(d) R_3 with d=200. Table 8 presents the comparison between *C-Rep* and *C-Rep-L* on synthetic data. The parameters using which the synthetic data is generated are also provided. We vary the relation size while keeping other parameters same. Table 9 presents the results for California road data. Here we consider the query $\mathcal{Q}_4 s$=R *Ov* R and R *Ra*(d) R. This finds road triples (rd_1, rd_2, rd_3) such that rd_1 overlaps with rd_2 and rd_2 is within distance d of r_3. We vary the parameter d while keeping the number of roads as 1m (sampled with probability 0.5). Once again we get the similar trends and *C-Rep-L* out-performs *C-Rep*.

10. CONCLUSIONS

In this paper we proposed novel approaches to handle multi-way spatial join queries on map-reduce platform. The proposed *Controlled-Replicate* framework is designed to minimize the communication among the cluster nodes. Using an experimental study over both synthetic and real-life California road data, we demonstrated that the proposed algorithms significantly improved the performance vis-a-vis naive methods. Though we presented our techniques in context of spatial data, we believe they are more general in applicability. We hence plan to look for use-cases other than spatial data. We also plan to look how we can use the proposed approaches for processing other classes of spatial queries e.g., nearest neighbor, containment etc.

11. REFERENCES

[1] Census 2000 Tiger/Line Data www.esri.com/data/download/census2000-tigerline.

[2] OpenMap Library http://openmap.bbn.com/.

[3] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.

[4] 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. In *SIGMOD*, 2010.

[5] T. Brinkhoff, H. P. Kriegal, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD*, 1993.

[6] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *SIGMOD*, 1994.

[7] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using r-trees. In *ICDE*, 1996.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.

[9] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *ICDE*, 2000.

[10] O. Günther. Efficient computation of spatial joins. In *ICDE*, 1993.

[11] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *SIGMOD*, 1994.

[12] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *SIGMOD*, pages 247–258, 1996.

[13] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10), 2012.

[14] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *SIGMOD*, 1999.

[15] N. Mamoulis and D. Papadias. Multiway spatial joins. *ACM Trans. Database Syst.*, 26(4), 2001.

[16] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.

[17] D. Papadias and D. Arkoumanis. Approximate processing of multiway spatial joins in very large databases. In *EDBT*, 2002.

[18] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, 1996.

[19] J. M. Patel and D. J. DeWitt. Clone join and shadow join: two parallel spatial join algorithms. In *ACM-GIS*, 2000.

[20] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[21] K. Wang, J. Han, B. Tu, J. Dai, W. Zhou, and X. Song. Accelerating spatial data processing with mapreduce. In *ICPADS*, 2010.

[22] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In *GCC*, 2009.

[23] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu. Sjmr: Parallelizing spatial join with mapreduce on clusters. In *CLUSTER*, 2009.