# Cruncher: Distributed In-Memory Processing for Location-Based Services

Ahmed S. Abdelhamid*, Mingjie Tang*, Ahmed M. Aly*, Ahmed R. Mahmood*, Thamir Qadah*, Walid G. Aref*, Saleh Basalamah‡

*Purdue University, West Lafayette, IN, USA ‡Umm Al-Qura University, Makkah, KSA

***Abstract*—Advances in location-based services (LBS) demand high-throughput processing of both static and streaming data. Recently, many systems have been introduced to support distributed main-memory processing to maximize the query throughput. However, these systems are not optimized for spatial data processing. In this demonstration, we showcase Cruncher, a distributed main-memory spatial data warehouse and streaming system. Cruncher extends Spark with adaptive query processing techniques for spatial data. Cruncher uses dynamic batch processing to distribute the queries and the data streams over commodity hardware according to an adaptive partitioning scheme. The batching technique also groups and orders the overlapping spatial queries to enable inter-query optimization. Both the data streams and the offline data share the same partitioning strategy that allows for data co-locality optimization. Furthermore, Cruncher uses an adaptive caching strategy to maintain the frequently-used location data in main memory. Cruncher maintains operational statistics to optimize query processing, data partitioning, and caching at runtime. We demonstrate two LBS applications over Cruncher using real datasets from OpenStreetMap and two synthetic data streams. We demonstrate that Cruncher achieves order(s) of magnitude throughput improvement over Spark when processing spatial data.**

## I. Introduction

The popularity of location-based services (LBS, for short) has resulted in an unprecedented increase in the volume of spatial information. In addition to the location attributes (e.g., longitude and latitude), the created data may include a temporal component (e.g., timestamp), and other application-driven attributes (e.g., check-in data, identifiers for moving objects, and associated textual content) [1]. Applications span a wide range of services, e.g., tracking moving objects, location-based advertisement, online-gaming, etc. Although these LBSs vary according to the nature of the underlying application, they share the need for high-throughput processing, low latency, support for adaptivity due to changes in location data distribution over time, and efficient utilization of the computing resources. This demands for the efficient processing of spatial data streams with high rates as well as huge amounts of static spatial data, e.g., OpenStreetMap. Moreover, the worldwide use of LBS applications requires processing of spatial queries at an unprecedented scale. For instance, LBSs are required to maintain information for tens if not hundreds of millions of users in addition to huge amounts of other service-associated data (e.g., maps and road networks), while processing millions of user requests and data updates per second.

Cloud computing platforms, where hardware cost is associated with usage rather than ownership, call for enhancing the query processing and storage efficiency. Furthermore, the dynamic nature of location data, especially spatial data streams and workloads, render the conventional optimize-then-execute model inefficient, and calls for adaptive query processing techniques, where statistics are collected to fine-tune the query processing and storage at runtime (e.g., see [2]).

One aspect that distinguishes LBSs is query complexity. In contrast to enterprise data applications, LBS queries are more sophisticated and can involve combinations of spatial, temporal, and relational operators, e.g., see [3], [4]. Some of these operators are expensive, e.g., k-nearest-neighbor (kNN) [5]. To address these challenges, various parallel and distributed systems are customized to handle location data, e.g., MD-Hbase [1], HadoopGIS [6], SpatialHadoop [7], Parallel Secondo [8], and Tornado [9]. These systems have a common goal; to store and query big spatial data over shared-nothing commodity machines. However, they suffer from disk bottlenecks, and provide no provisions for adaptive query processing.

Recently, the significant drop in main-memory cost has initiated a wave of main-memory distributed processing systems. Spark [10] is an exemplification of such computing paradigm. Spark provides a shared-memory abstraction using Resilient Distributed Datasets (RDDs, for short) [11]. RDDs are immutable and support only coarse-grained operations (referred to as transformations). RDDs keep the history of transformations (referred to as Lineage) for fault tolerance. RDDs are lazily evaluated and ephemeral. An RDD transformation is only computed upon data access (referred to as Actions) and data is kept in memory only upon deliberate request. In addition, Spark supports near-real-time data stream processing through small batches represented as RDDs (referred to as Discretized Streams) [12]. However, Spark is not optimized for spatial data processing and makes no assumptions about the underlying data or query types.

This demonstration presents Cruncher, a distributed spatial data warehouse and streaming system. Cruncher provides high-throughput processing of online and offline spatial data. Cruncher extends Spark with adaptive query processing techniques. Originally, Spark processes data stream records in order of arrival. However, processing a batch of data elements or queries offers an opportunity for optimization and renders the fixed batch content ordering sub-optimal. Hence, Cruncher introduces a new batching technique, where the system dynamically changes the batch content ordering to update the RDDs efficiently. In addition, processing a batch of multiple queries offers an opportunity for multi-query optimization, and hence Cruncher introduces an inter-query optimization technique for range and kNN queries. Furthermore, Spark speeds-up the data processing by partitioning the data in main memory.

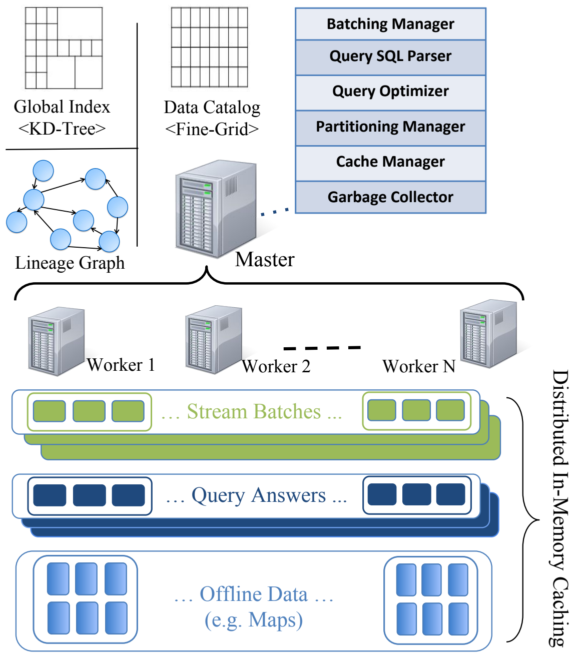


Fig. 1. An overview of Cruncher.

However, static partitioning of spatial data over a distributed memory is not robust in case of dynamic workloads and dynamic data distributions. Hence, Cruncher uses an adaptive partitioning technique that recognizes the query/data hot spots, and incrementally updates the data partitioning to minimize redundant processing. Finally, Cruncher introduces a garbage collector to remove outdated and obsolete RDDs from the main memory. Cruncher uses three type of runtime statistics; a global index of location data partitions, a grid with fine granularity to maintain count statistics for the location-based data and queries, and a lineage graph that tracks the RDD transformations. We demonstrate Cruncher's capabilities using data streams of moving objects from BerlinMod [5] and real static datasets from OpenStreetMap [13].

## II. Overview of Cruncher

### A. Supported Features

Below, we summarize the main features of Cruncher:

**High Throughput Processing.** Cruncher uses dynamic batching and inter-query optimization to achieve high query throughput given the underlying resources. Cruncher achieves orders of magnitude throughput improvement over Spark.

**Online and Offline Processing.** In addition to offline data (e.g., maps), Cruncher handles three types of location data streams, namely, a user queries stream, a data updates stream (to be applied to the offline data), and application data streams. Cruncher can process queries with relational and spatial predicates against combinations of all these types of data sources.

**Adaptive Main-Memory Data Partitioning.** RDDs do not make any assumptions about the properties of the underlying data nor the incoming queries. In contrast, Cruncher adaptively update the RDD partitioning at runtime to cope with the changes in the query workload and data distribution, and hence consistently maximize the query throughput.

**Efficient Memory Evacuation Policy.** Because Cruncher relies on RDD transformations for query processing and data updates, multiple RDDs may co-exist in memory carrying data for the same spatial range fully or partially. For efficient memory use, only single data copy should be maintained per spatial region that reflects the most recent updates. Cruncher utilizes an efficient garbage collector with spatial awareness to eliminate in-memory duplicates.

**Interactive Map-Assisted GUI.** Cruncher supports an interactive GUI that extends Apache Zeppelin [14] to support SQL-like and Map-UI querying.

**Light-Weight Fault Tolerance.** Cruncher extends the lineage-based fault tolerance mechanism of the RDD model [11]. By associating runtime statistics with the RDD transformations, Cruncher persists the updated RDDs efficiently on disk.

### B. Supported Queries

Cruncher aims to support queries that include both spatial and relational predicates, where multiple spatial predicates can appear within a single query. The supported spatial predicates include *Range, kNN Select, kNN join, and Spatial Join*. Cruncher also supports *temporal and textual predicates*. Queries can run against online streams or offline (i.e., static) data, and can be snapshot or continuous. Examples of the supported queries are presented in Section IV.

### C. Data Model

An LBS can store data about stationary as well as moving objects and queries. The following updates are continuously received by Cruncher: 1) periodic updates for the locations of the moving objects/queries and their associated data, e.g., the time of the update, the text associated with the new location, e.g., tweets from the new location, etc., 2) service updates for the stationary data, and 3) queries that include spatial, temporal and textual predicates. The stationary and moving objects have the format of:{object-identifier, location, timestamp, relational data, free text}. The queries can be represented in SQL, from which the following format is extracted: {query-identifier, timestamp, location, predicate-list}.

## III. In-Memory Processing in Cruncher

Cruncher employs a set of techniques that achieve efficient distributed main-memory processing of spatial data. This section highlights each of these techniques.

### A. Adaptive Partitioning with On-Demand Indexing

Cruncher dynamically partitions in-memory data to redistribute data over the cluster. The Objective is to repartition the RDDs based on the query workload and data updates, such that a query operates on a minimal set of data required to retrieve its answer. Cruncher's Partitioning Manager extends our work in [2]. As in [2], two global indices are maintained. Refer to Fig. 1. A k-d tree index represents the current data partitioning scheme, and a fine-grained grid maintains the count of data points and queries at each grid cell. The

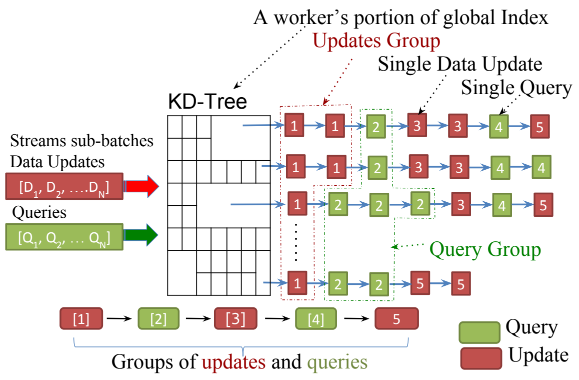


Fig. 2. Dynamic Batch Processing.

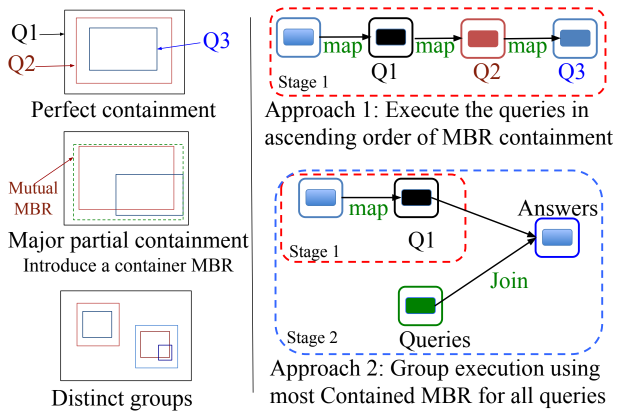

Q1
Q2
Q3
Perfect containment
Mutual MBR
Major partial containment
Introduce a container MBR
Distinct groups
map
Stage 1
Approach 1: Execute the queries in ascending order of MBR containment
Answers
Join
Stage 2
Queries
Approach 2: Group execution using most Contained MBR for all queries

Fig. 3. Multi-query Optimization.

repartitioning is incrementally triggered based on a cost model that minimizes redundant data processing. The cost model integrates the number of points and queries per partition. In Cruncher, we introduce two extensions. First, we make the cost model aware of the data updates. Second, when applying a batch of designated queries on a particular data partition, we consider the nature of the underlying operators. For example, in the case of extensive use of kNN operators in a certain partition, we can dynamically build a suitable index for this partition to facilitate the execution of the kNN operators within this partition. Subsequently, the index can be invalidated and removed upon data updates or partitioning changes, or simply to preserve memory when the index is not needed anymore.

## B. *Dynamic Batch Processing*

Cruncher partitions the incoming batches of data updates and queries into small sub-batches according to the data partitioning scheme. Each sub-batch is sent to the machine responsible for processing and caching the data for its spatial region. This partitioning is dynamic and can be different for every batch as discussed in Section III-A. Furthermore, Cruncher groups and sorts the sub-batches for processing. Recall that in the RDD model, every update triggers the creation of a new RDD. To handle frequent updates common in LBSs, Cruncher minimizes the number of transformations required to update the data in order to reduce the writing overhead. Cruncher supports two modes of operation: consistent and greedy. In the former, updates and queries are grouped. The correctness of evaluation is achieved by guaranteeing that a query will never process an item, where an update is available with a timestamp proceeding that of the query, until the update is applied first. Fig. 2 shows how a batch of queries/updates is sorted and grouped as a series of transformations. The grouping and sorting are based on the timestamps and the spatial regions of the updates/queries. In the greedy mode, the sub-batch is divided into two transformations only (i.e., queries vs. updates). The queries are applied first, and then the updates. This mode is suitable for LBS applications with relaxed correctness criteria but that are sensitive to latency. Minimizing the number of RDD creations leads to shorter lineage, and hence reduces the overhead during crash recovery.

## C. *Multi-Query Optimization*

Batch query processing creates opportunities for multi-query optimization. Cruncher applies location-specific inter-query optimization, e.g., based on the containment in minimum bounding rectangles (MBRs, for short). Consider a location-based query, say $Q1$, that is MBR-contained within another query's MBR, say $Q2$'s. Cruncher sorts the queries within a batch based on MBR containment. Refer to Fig. 3. 3 cases for sorting a query group based on their MBRs are illustrated. The queries are executed using one of two approaches. First, queries can be executed sequentially as a series of transformations in descending order of MBR size. Alternatively, we can join the predicates with the subset of data that includes all these predicates. In other words, we apply one transformation to get the data of the biggest MBR, and then join with all the predicates. In the first approach, an RDD is defined per query, but in the latter approach, we have one and only one RDD, where each tuple is tagged with the satisfying query.

## D. *Distributed Workload-aware Caching*

Continuously applying RDD transformations can result in multiple copies of the same data in main memory (i.e., in different RDDs). Recall that updating an RDD creates a new RDD with the update. Also, applying a spatial query on an RDD, creates a new RDD for the query answer holding a subset of the original RDD. To avoid these multiple copies, Cruncher applies a caching mechanism that: 1) increases the memory hit-ratio by keeping the frequently accessed data in main memory, and 2) reduces the replication factor of data in memory. Fig. 4 shows how a spatial index (a grid with fine granularity) maintains access-counters in the grid cells as well as coverage relationships among the RDDs and the grid cells. For each grid cell, Cruncher keeps a counter of usage plus the time of last access. This information is useful for LRU or ARC cache replacement policies when the data does not fit in main memory. In addition, the Garbage Collector uses the coverage relationship to keep in memory only the most updated RDDs for each spatial region and evacuate outdated RDDs.

## E. *Fault Tolerance*

Cruncher relies on the lineage-based fault-tolerance mechanism of RDDs. However, applying many transformations on the RDDs results in long lineage chains. It is vital to keep the lineage graph manageable by forcing periodic persistance of RDDs (i.e., saving them to disk) when necessary, otherwise the re-computations of the RDDs in case of failure can be

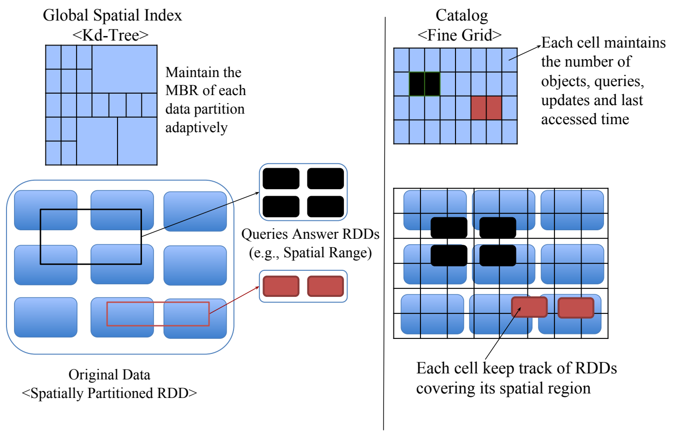

Global Spatial Index
<Kd-Tree>
Maintain the MBR of each data partition adaptively
Queries Answer RDDs (e.g., Spatial Range)
Original Data
<Spatially Partitioned RDD>
Catalog
<Fine Grid>
Each cell maintains the number of objects, queries, updates and last accessed time
Each cell keep track of RDDs covering its spatial region

Fig. 4. Spatial-aware Caching.

more expensive than reading from disk. Cruncher uses a simple, yet effective, technique to maintain a manageable lineage chain. Cruncher keeps track of the processing time of all the transformations. When the total processing time of a sequence of transformations exceeds the expected reading time from disk, checkpointing is triggered to save the RDD to disk. Observe that Cruncher keeps track of the computed transformations only. The transformations yet to be computed are not counted.

## IV. Demo Scenario

We demonstrate Cruncher's capabilities using two applications, where we use a real dataset of points of interest from OpenStreetMap [13] and synthetic datasets of moving objects from the BerlinMod Benchmark [5]. We append a textual description to each moving object. We generate a synthetic data stream that simulates offers (e.g., coupons) made by the restaurants in the OpenStreetMap dataset.

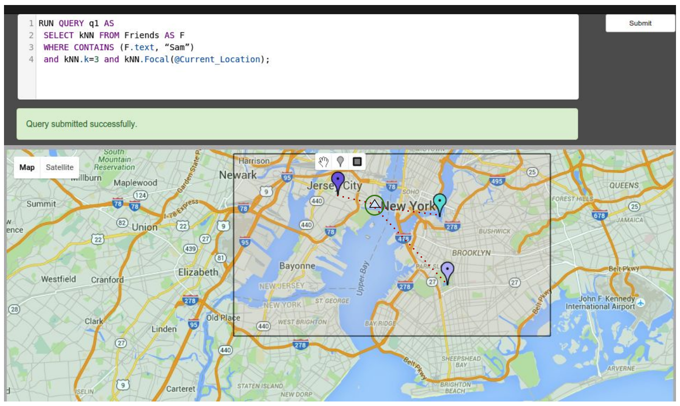
**Online Data Processing:** In this scenario, the user locates her nearby friends who have certain text associated to them. This query runs against a stream of moving objects, and can be expressed as follows:

```
RUN QUERY q1 AS
SELECT kNN FROM Friends AS F
WHERE CONTAINS (F.text, ‘Sam’)
and kNN.k=3 and kNN.Focal(@Current_Location);
```

**Online and Offline Data Processing:** In this scenario, the user gets notified of offers (e.g., coupons, sales) applicable to the restaurants that are inside a specific spatial region. This is a continuous query that requires hybrid processing of an online stream (i.e., offers) and offline data in the map. The query can be expressed as follows:

```
REGISTER QUERY q2 AS
SELECT * FROM OSM_Data AS O, OFFERS AS F
WHERE INSIDE(O, @Spatial_Range)
and CONTAINS(F.text, ‘Coupon’, ‘Sale’)
and OVERLAPS(O.text, F.text)
and F.type = ‘Restaurant’;
```

We will show the performance of Crusher under different rates of online data streams and user queries. We will use different batch sizes for processing various combinations of the above queries. We will visualize how the global data partitioning scheme adapts given the change in the query workload and data updates. We will compare the throughput of Cruncher against original Spark. Conducted experiments show how Cruncher achieves orders of magnitude improvement in query throughput.



```
RUN QUERY q1 AS
 SELECT kNN FROM Friends AS F
 WHERE CONTAINS (F.text, “Sam”)
 and kNN.k=3 and kNN.Focal(@Current_Location);
```

Query submitted successfully.

Fig. 5. Cruncher SQL and Map User Interface.

## Acknowledgment

This research was supported in part by National Science Foundation under Grant Number IIS 1117766.

## References

[1] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, “MD-HBase: A scalable multi-dimensional data infrastructure for location aware services,” in *IEEE MDM*, 2011.

[2] A. M. Aly, A. S. Abdelhamid, A. R. Mahmood, W. G. Aref, M. S. Hassan, H. Elmeleegy, and M. Ouzzani, “A demonstration of aqwa: Adaptive query-workload-aware partitioning of big spatial data,” in *VLDB*, 2015.

[3] A. M. Aly, W. G. Aref, and M. Ouzzani, “Spatial queries with k-nearest-neighbor and relational predicates,” in *SIGSPATIAL*, 2015.

[4] ——, “Spatial queries with two knn predicates,” in *VLDB*, 2012.

[5] C. Düntgen, T. Behr, and R. H. Güting, “Berlinmod: a benchmark for moving object databases,” *VLDB J.*, 2009.

[6] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz, “Hadoop-GIS: A high performance spatial data warehousing system over mapreduce,” *PVLDB*, vol. 6, no. 11, 2013.

[7] A. Eldawy and M. F. Mokbel, “SpatialHadoop: A mapreduce framework for spatial data,” in *ICDE*, 2015.

[8] J. Lu and R. H. Guting, “Parallel secondo: A practical system for large-scale processing of moving objects,” in *ICDE*, 2014.

[9] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, and S. B. Walid G. Aref, “Tornado: A distributed spatio-textual stream processing system,” in *VLDB*, 2015.

[10] “Apache Spark,” https://spark.apache.org/, 2015.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *NSDI*, 2012.

[12] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *SOSP*, 2013.

[13] “OpenStreetMap,” http://www.openstreetmap.org/, 2015.

[14] “Apache Zeppelin,” https://zeppelin.incubator.apache.org, 2015.