ISP: Large-Scale In-memory Spatial Data Processing System (Demo Paper)

Simin You
Dept. of Computer Science
CUNY Graduate Center
New York, NY 10016
syou@gc.cuny.edu

Jianting Zhang
Dept. of Computer Science
The City College of New York
New York, New York 10031
jzhang@cs.ccny.cuny.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK, USA
ggruenwald@ou.edu

ABSTRACT

Huge amount of spatial data such as GPS locations is being generated everyday, which brings big challenges of efficient spatial data processing. Many existing big spatial data processing techniques are mostly based on disk-resident systems. They have not fully taken advantages of modern hardware, such as large main memory capacities and multi-core processors. In this paper, we demonstrate our ISP system for in-memory processing of large-scale spatial data in distributed multi-core computing nodes. ISP is built on top of the open source Impala system, a leading Massively Parallel Processing (MPP) SQL engine, with two signficant extensions. First, while Impala is designed to process relational data and does not support spatial queries, ISP supports spatial SQL query syntax at the front end and is able to process the spatial queries at the back end. Second, while Impala currently supports neither indexed joins nor parallel joins on multi-core machines for non-equality joins, ISP provides on-the-fly parallel spatial indexing and query processing modules. We have performed experiments for a case study of point-in-polygon test based spatial joins. Using real data for a point-in-polygon based spatial join, experiments have shown that ISP on a two-node mini-cluster is 6.4X time faster than PostgreSQL/PostGIS. ISP is also 10X faster than Hadoop-GIS, a big spatial data processing solution built on top of Hadoop/Hive, on a 10-node Amazon EC2 cloud cluster. With proper setting of parameters of distributed systems, ISP also scales well. We will demonstrate ISP using both an EC2 cloud cluster and an in-house small cluster to conference participants.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications – *Spatial Databases and GIS*

General Terms

Performance, Design, Experimentation.

Keywords

Big Spatial Data, Spatial Join

1. INTRODUCTION

The Increasingly popular mobile devices have generated tremendous amount of location data, such as GPS and wifi locations. Efficiently managing big spatial data in a scalable way is technically challenging. Emerging big data technologies such as MapReduce/Hadoop have become *de facto* standard for data storage and processing. In order to process big spatial data efficiently, several extensions have been developed to support spatial data processing. Hadoop-GIS [1] enables spatial functionalities on existing Hadoop systems by extending Hive [2]. SpatialHadoop [3] implements a set of spatial operations that can be used in Map/Reduce jobs. ESRI GIS Tools for Hadoop [4] is another open source project that enables spatial data processing on Hadoop and implements spatial functions in Hive [2].

As memory is getting significantly cheaper and computers are increasingly equipped with large memory capacities, there are considerable research and application interests in processing large-scale data in memory to reduce disk I/O bottlenecks and achieve better performance. Newer generations of big data systems such as Apache Spark [5] and Cloudera Impala [6], although still support HDFS (Hadoop Distributed File System) for persistent storage, are able to take advantage of the large memory capacities and achieve higher performance. However, we are not aware of existing systems that use or extend in-memory big data systems for large-scale spatial data processing. In-memory spatial processing techniques, such as in-memory R-Tree indexing for speeding up topological relationship test [7] and the in-memory data structure in TOUCH for efficient spatial join [8] have been recently proposed. Our previous work on grid-file and R-Tree based spatial indexing and query processing on Graphics Processing Units (GPUs) can also be considered as in-memory techniques [9][10], where GPU-specific characteristics, such as data parallelisms, control divergence and coalesced memory accesses, are carefully considered. However, these techniques are designed for single-node computation and cannot easily scale out to distributed machines to process larger-scale data.

To achieve both efficiency and scalability in large-scale spatial data processing, we have designed and implemented the ISP prototype system by significantly extending Impala [6], a leading open source Massively Parallel Processing (MPP) SQL engine for relational data. ISP extends Impala's front end to parse spatial SQL syntax and generate logic and physical query plans. Although Impala can efficiently utilize multi-core CPUs for multi-threaded disk I/Os and scans and aggregations on relational data, currently it has limited support for joins where only hash joins can effectively utilize multiple CPU cores. As such, we have significantly modified Impala's backend by providing on-the-fly spatial indexing and supporting spatial joins on multi-core CPUs in parallel. Compared with Hadoop-GIS and SpatialHadoop, ISP is built on top of a newer generation big data system and is able to achieve higher performance by taking advantage of efficient in-memory processing. Advanced system infrastructure support from Impala, including Just-In-Time (JIT) compilation of complex expressions in SQL using the open source LLVM compiler [11], is also critical in achieving the high-performance. For the rest of the paper, we will first introduce the system architecture of ISP and its implementation details (Section 2). Demonstration plans, including experiment setup, case study data and observed

performance as well as comparisons with a popular spatial database (PostgreSQL/PostGIS) and a big spatial data processing system (Hadoop-GIS) are provided in Section 3.

2. SYSTEM ARCHITECTURE

2.1 Impala for Efficient Big Data Processing

Traditional MapReduce jobs usually incur large amounts disk IO to store intermediate results. Cloudera Impala [6], as a new generation big data system, is designed to take advantage of large memory capacities to significantly reduce disk IO overheads. As a component in MapReduce/Hadoop ecosystem, Impala supports HDFS and can read data in different formats. Similar to Hive and Shark[12], Impala has a Java-based front end to parse SQL statements and generate logical query plans with rule-based optimizations. By consulting Hive *metastore*, Impala front end is able to generate efficient physical query plans by incorporating cost-based optimizations. A query plan is then sent to an Impala backend instance that works as the master. The master coordinates with a set of worker Impala instances to process a query in a distributed manner. The query results are gathered by the master and sent back to clients (e.g., a shell program that initiates the query). Different from traditional MapReduce/Hadoop systems that write intermediate query results to HDFS, an Impala instance processes all the computation that is being assigned to it completely in memory without touching HDFS except for explicit outputting.

Clearly Impala trades fault tolerance with efficiency which can be well justified in systems that failure rates are low [13]. As one of the few open source big data systems with a C/C++ based execution engine, Impala is ideal to serve as the base for further extensions when performance is critical. In particular, as currently Java does not support exploiting SIMD (Single Instruction Multiple Data) computing power [14] on either CPUs or GPUs, C/C++ language interfaces might be the most viable option to effectively utilize hardware accelerations. While the current version of ISP primarily focuses on in-memory spatial query processing using multiple processing cores on CPUs, we plan to exploit SIMD computing power for higher performance on new commodity parallel hardware, including GPUs and Intel Xeon Phi accelerators in the future.

To our knowledge, currently Impala is the only open source big data system that supports Just-In-Time (JIT) compilation using the leading open source LLVM compiler. LLVM supports not only x86/64 hardware architectures but also many new ones, such as ARM and Nvidia PTX, among others [11]. JIT is essential for high-performance query processing on modern hardware [15] by supporting various types of dynamic code optimizations, such as compiling complex expressions and seamlessly integrating User Defined Functions (UDFs) with native host machine code. Despite that currently Impala has limited support for joins (more discussions next), its advanced infrastructure and efficient implementations of parallel scans and aggregations make it attractive to serve as the base for extensions.

Our ISP system is built on top of Impala and naturally inherits many advanced features of Impala, including supporting SQL interface, efficient multi-threaded disk IO to HDFS, and in-memory query processing. In order to extend Impala for high-performance spatial query processing, we have made three major extensions. First, we modify the Abstract Syntax Tree (AST) module of Impala front end to support spatial query syntax. Second, we have developed a set of UDFs for testing spatial relationships (by wrapping the popular GEOS package [16]) for the Impala backend. Finally and most significantly, as Impala currently does not support non-equality joins using multi-core CPUs natively, we have developed both a spatial indexing module and a spatial query processing module to support parallel spatial query processing in Impala natively, which is crucial to fully utilize multi-core CPUs to achieve the desired high performance. The architecture of ISP is shown in Fig. 1.

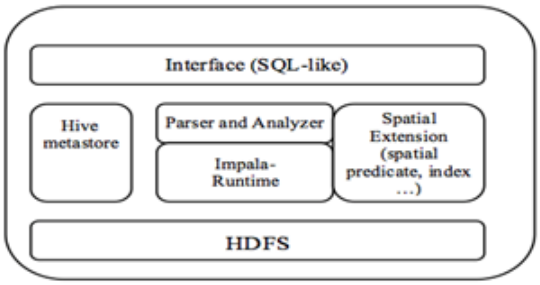

Fig. 1 Overview of ISP Architecture

2.2 Spatial Extensions on Impala

Currently ISP represents geometric object using the simple and popular Well-Known Text (WKT) format and store the data as strings, which is natively supported by Impala. Although it seems to be inefficient when compared to existing spatial databases that store geometric data in a binary format, our experiments have shown that parsing WKT incurs relatively low overheads when compared with expensive geometric operations and is acceptable. While we plan to extend Impala to support binary spatial data natively in the future, storing geometric object as strings make it possible to interoperable with Hadoop based systems. ISP uses GEOS library to parse strings in WKT format and reconstruct in-memory geometric objects on the fly.

ISP is designed to support spatial queries through UDFs which is well-supported by Impala. Different from Java-based big data systems that require a UDF provide a native Java interface before it can be invoked, a UDF in Impala can be compiled into Intermediate Representation (IR) code that can be written in any langue, in addition to calling APIs in a shared library. Since we want to extensively explore JIT for high performance, ISP has adopted the IR approach. We have developed a few spatial UDFs to test spatial relations between two geometric objects that are provided to UDFs as strings in WKT format. This is realized by wrapping around the corresponding functions in GEOS, such as *ST_Intersects* and *ST_Within*. While calling external APIs within IR code in the current version makes it impossible to completely seamlessly integrate spatial UDF code with Impala system code (as LLVM-based JIT compilation cannot be extended to API internals), the design makes it possible for future expansions.

Currently our spatial UDFs are simply wrapper functions using GEOS library without SIMD support. However, we plan to rewrite such functions that can utilize increasingly powerful Vector Processing Units (VPUs) on multi-core CPU, e.g., the 4-way SSE, 8-way AVX/AVX2 and 16-way AVX-512 (on Intel Xeon Phi accelerators and forthcoming Intel Xeon processors). Our standalone experiments have demonstrated the potentials of using SIMD computing power for geospatial operations [17]. When native spatial UDFs with SIMD parallel computing are fully compiled into IR code and seamlessly integrated with Impala system code in the next version, we expect ISP can achieve even higher performance.

Once spatial UDFs are registered with Impala, they can be used to support both "single-sided" ad-hoc spatial queries and "double-sided" spatial joins, through different mechanisms. For "single-sided" ad-hoc spatial queries, one or more geometric objects are provided in WKT format in a SQL expression. The geometric objects are used to query against a table stored in HDFS. Since the geometric object stored in the table is also in WKT format and is typically non-indexed, a full table scan with on-the-fly expression evaluations of the selection criteria to each and every of the tuples in the table is appropriate. This is exactly many big data systems are designed for. No additional extensions are needed for Impala to support such "single-sided" ad-hoc spatial queries other than providing spatial UDFs. However, it is much more involved for ISP to support "double-sided" spatial joins on top of Impala where indexing is necessary to avoid a full cross join. We note that, although full cross join on spatial data is naturally supported in Impala by using spatial UDFs, the complexly is O(*m***n*) where *m* and *n* are the numbers of tuples in the two tables being joined. While the approach is embarrassingly parallelizable and is suitable for massively parallel systems, it scales poorly and is impractical for large-scale data. Currently Impala can only support equality joins on relational data efficiently by using well-established hash join techniques. As such, the most signficant technical contribution of ISP is to provide indexed spatial joins natively within Impala framework. Our solution, although still limited in a sense and has room for future improvements, as demonstrated in Section 3, has achieved signficant speedups over alternatives. The implementation details are provided in the next subsection (Section 2.3).

We would like to compare ISP with the approaches adopted in Hadoop-GIS [1] and SpatialHadoop [3] to help understand the achieved high performance to be reported in Section 3.2. Different from SpatialHadoop that requires users to write MapReduce jobs for spatial queries, which is flexible but less user-friendly, both ISP and Hadoop-GIS accept spatially extended SQL statements. However, unlike ISP that evaluates queries in-memory, the Hadoop-GIS front end (i.e., Hive-SP) compiles a SQL query into multiple sequential MapReduce jobs. The results of spatial UDF evaluations are written to HDFS before they are combined with other inputs to perform the rest of the query, which now becomes a regular SQL query in Hive. While caching the intermediate spatial and non-spatial query results in RAM disks can certainly improve system performance to a certain extent, excessive redundant disk IO will hurt the overall performance significantly in this case. Furthermore, as both Hadoop-GIS and SpatialHadoop are built on top of traditional MapReduce/Hadoop systems, they use multi-core machines as multiple virtualized machines with single processors to process Map/Reduce jobs without being aware of data and computing locality at multiple levels. In contrast, ISP is able to exploit the raw computing power of multi-core machines and optimize system performance using native parallel programming tools (e.g., OpenMP and Intel TBB) for better performance.

2.3 Implementation Details of Spatial Joins

While spatial joins involve two tables stored in HDFS in ISP, we observe that very often these two tables are asymmetric with respect to the numbers of tuples. For example, in our taxi trip data management applications [9], while the numbers of pickup and drop-off locations increase quickly (half a million a day in New York City – NYC), the underlying urban infrastructure data such as road network and census tract data are "small" in volume. As such, it is possible to broadcast the "small" table to all computing node and index them on the fly so that tuples in the "big" table can probe the index structure in parallel. The strategy is different from traditional spatial databases that typically choose to index the "big" table and use all the tuples in the "small" table to query the index in a spatial join. This is because offline indexing time is not counted in such as a scenario. However, when both indexing and querying time are considered and a query is distributed among K computing nodes, our preliminary complexity analysis has shown that, on-the-fly indexing the broadcast "small" table on each node and querying 1/K tuples of "big" table on the node incurs lower process time than the other way around.

When executing a SQL query on a computing node, starting from the root of an AST, expressions that are associated with AST nodes are evaluated top-down using the appropriate data partitions that are assigned to the Impala instance on a computing node. When joining two tables based on an UDF, the AST node corresponding to the join has two child nodes with necessary information of the two tables. Starting from here, Impala requests blocks of tuples from the two tables in an iterative manner to process the join in batches. The implementation of our spatial join extension works as follows. First, we iteratively retrieve the geometry columns of tuples of the "small" table and build an appropriate spatial index for all the tuples in the "small table". Retrieving the "small" table from HDFS can be efficiently done using multi-threaded I/O supported by Impala. Second, we iterate through all the blocks that are assigned to an Impala instance sequentially to perform the spatial join. For each block, we use OpenMP to parallelize tuple evaluations within a block. Non-spatial sub-expressions are evaluated first before the spatial UDF is evaluated. Tuple pairs satisfy the criteria defined in the WHERE clause (including one or more spatial UDFs and other non-spatial sub-expressions) are written to a tuple buffer before they are sent to upper level AST node for subsequent processing in blocks, e.g., projection and aggregations and upper level SQL statements if a sub-query is being processed. At any time during the process of executing a SQL query, no more than three blocks of tuples (the two input blocks and the output buffer block) are stored in main memory for an AST node, unless additional dynamic structures are purposely maintained (e.g., hash tables for relational indexing).

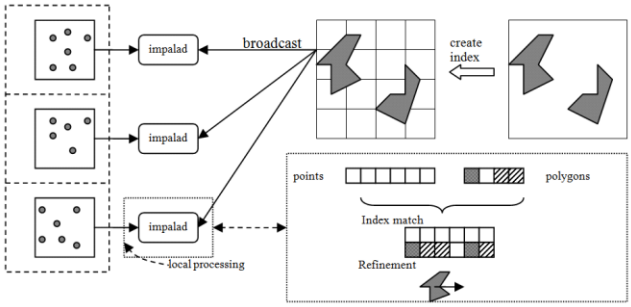

Fig. 2 Point-in-polygon based Spatial Join Processing in ISP

ISP takes advantage of Impala's carefully designed framework for SQL execution. Instead of pairing tuples from two joining tables in a cross product manner (which is the default), as discussed above, ISP builds a grid indexing structure for the "small" table. By using the "*parallel for*" OpenMP directive, tuples in a block of the "big" table are assigned to multiple threads for parallel evaluating the operation represented in the AST node that is being traversed. As discussed before, UDFs and sub-expressions in the operations can be JIT compiled to host machine code if the corresponding IR codes are available. Fig. 2 shows an example on point-in-polygon test based spatial join to illustrate the design

of spatial join extensions to Impala which is a unique contribution from ISP.

3. DEMOSTRATION

In the section, we will demonstrate a use case of point-in-polygon test based spatial join in ISP. Consider we want to find the zone (*polygon*) that each GPS location (*point*) falls within, the query can be expressed as an SQL statement as following:

```
SELECT point.id, polygon.id
FROM point SPATIAL JOIN polygon
WHERE ST_WITHIN(point.geom, polygon.geom)
```

3.1 Setup

For the point dataset, we used subsets of NYC Taxi GPS trip data [9]. We have prepared two datasets from the original dataset for the demonstration, including a small dataset (~2 million points, named *point_small*) and a large dataset (~28 million points, named *point_large*). The polygon dataset we used is from NYC PLUTO tax lot dataset [18]. We extracted a large dataset with 824,811 polygons (in all boroughs excluding Staten Island) and a small dataset (which covers only Manhattan) with 43,252 polygons from PLUTO as the testing polygon datasets. They are called *polygon_large* and *polygon_small*, respectively.

In this demonstration, we prepared three experimental configurations. The first one is a SGI Octane III mini-cluster that has two nodes, in which each node has dual quad-core CPUs and 48GB memory. Each node in the mini-cluster runs as a Hadoop datanode as well as an Impala instance. Participant can try ISP on the in-house mini-cluster in an unrestricted manner. We also installed PostgreSQL 9.2.3 on one of the two mini-cluster nodes for comparison purposes. The third configuration is an Amazon EC2 cluster with up to 10 *c1.xlarge* instances. Each instance has 8 vCPUs and 7 GB memory. We have installed and configured ISP on the cloud cluster and will use it to compare with HadoopGIS.

3.2 Performance and Scalability

We imported *point_large* and *polygon_small* datasets into PostgreSQL with built-in spatial indices created after data import. A point-in-polygon spatial join query took about 1.7 hours to finish on one node of the two-node mini-cluster. The poor performance is mainly due to sequential execution in PostgreSQL/PostGIS on a single CPU core, which significantly underutilizes available hardware resources. The same query performed on our ISP using both nodes of the mini-cluster took 16 minutes where all CPU cores were utilized. In addition to the 6.4X speedup, another advantage of ISP over PostgreSQL is that ISP requires no additional performance tuning. Furthermore, ISP can process data stored in HDFS directly without involving expensive import/export steps as in PostgreSQL/PostGIS.

We also compared with HadoopGIS [3], a big spatial data query processing system. Two datasets, *point_small* and *polygon_large*, are used as the testing datasets. We allocated 10 c1.xlarge nodes for both HadoopGIS and our ISP. We measured end-to-end query times for both systems. The experiment showed that HadoopGIS took 6 minutes and our ISP took only 37 seconds where a nearly 10X speedup has been observed.

To test ISP's scalability, we vary the number of c1.xlarge nodes (instances) on the Amazon EC2 cluster from 4 to 10 for the same point-in-poly test based spatial join. The runtimes are plotted against the numbers of instances in Fig. 4. We can see that ISP scales linearly up to 8 nodes but does not scale further beyond that. Careful investigation suggests that this is due to the scheduling policy imposed by Impala where an Impala instance participate a join only if it has a local partition of the "big" table. Using a HDFS data block size of 16MB, there are only 7 data blocks of the point dataset in HDFS and thus using more nodes beyond 7 will not help. Larger "big" table volume will naturally have larger number of data blocks which will scale out further on ISP. The finding can also be used to guide setting proper HDFS parameter and/or deciding cloud computing resource demands.

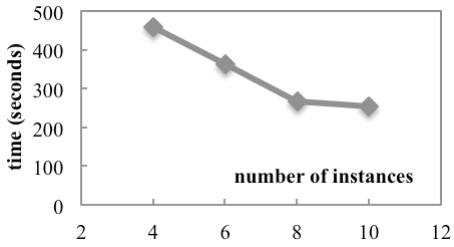

Fig. 3 Plot of Runtimes Against Number of Instances

4. SUMMARY AND CONCLUSTION

We have developed ISP, a large-scale in-memory system based on Impala with spatial extensions that directly operates on HDFS. Using a case study of point-in-polygon test based spatial join, we demonstrate that ISP significantly outperforms both a traditional spatial database (6.4X over PostgreSQL/PostGIS) and a MapReduce/Hadoop based spatial big data system (10X over Hadoop-GIS). We have provided plans to demonstrate the efficiency and scalability of ISP to the conference participants.

5. REFERENCES

[1] A. Eldawy and M. Mokbel, "A demonstration of Spatialhadoop: an efficient mapreduce framework for spatial data," in Proc. VLDB, 6(2), 1230-1233, 2013.

[2] Apache, Hive, https://hive.apache.org/

[3] A. Aji et al, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce," in Proc. VLDB, 6(11), 1009-1020, 2013.

[4] ESRI, http://esri.github.io/gis-tools-for-hadoop/.

[5] Apache, Spark, http://spark.apache.org/.

[6] Cloudera, Impala, http://impala.io/.

[7] Y. Hu et al, "Topological relationship query processing for complex regions in Oracle Spatial," in Proc. ACM-GIS, 2012.

[8] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan and A. Ailamaki, "TOUCH: in-memory spatial join by hierarchical data-oriented partitioning," in Proc. ACM SIGMOD Conference, 2013.

[9] J. Zhang, S. You and L. Gruenwald, "Parallel Online Spatial and Temporal Aggregations on Multi-core CPUs and Many-Core GPUs," Information Systems, vol. 4, p. 134–154, 2014.

[10] J. Zhang and S. You, "GPU-based Spatial Indexing and Query Processing Using R-Trees," in Proc. ACM-GIS, 2013.

[11] LLVM, The LLVM Compiler Infrastructure, http://llvm.org/.

[12] Berkeley, Spark. http://shark.cs.berkeley.edu/

[13] K. A. Kumar et al. "Optimization Techniques for "Scaling Down" Hadoop on Multi-Core,Shared-Memory Systems," in Proc. EDBT, 2014.

[14] J. Parri, et al, "Returning Control to the Programmer: SIMD Intrinsics for Virtual Machines," ACM Queue, 9(2), 30-37, 2011.

[15] C. Koch, "Abstraction without Regret in Database Systems Building: a Manifesto," IEEE Data Engineering Bulletin, 37(1), 70-79, 2014.

[16] OSGEO, GEOS - Geometry Engine, Open Source, http://trac.osgeo.org/geos/.

[17] J. Zhang and S. You, "Large-Scale Geospatial Processing on Multi-Core and Many-Core Processors: Evaluations on CPUs, GPUs and MICs," CoRR, vol. abs/1403.0802, 2014.

[18] NYCDCP, BYTES of the BIG APPLE, http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml#pluto.