A Scalable Indexing Technique and Implementation for Big Spatial Data
A Scalable Indexing Technique and Implementation for Big Spatial Data

Table of Contents

1 Executive Summary ............................................................................................................................................ 4
2 Introduction.......................................................................................................................................................... 4
  2.1 TST Business Approach......................................................................................................................... 4
  2.2 Current Customer Examples..................................................................................................................... 5
3 Current Capabilities and Future Plans......................................................................................................... 5
  3.1 iSpatial – Geo Visualization Framework ................................................................................................. 5
  3.2 Ubiquity – Mobile Application Framework ............................................................................................ 8
  3.3 Future Plans................................................................................................................................................ 10
4 MongoDB Architecture, Organization, and Data Structures................................................................. 11
  4.1 Software Implementation Overview .......................................................................................................... 13
  4.2 Data Structures.......................................................................................................................................... 16
  4.3 Spatial Implementation and Limitations..................................................................................................... 18
5 MongoDB Spatial Improvements................................................................................................................ 20
  5.1 R-Tree Indexing Algorithms and Implementation ..................................................................................... 21
  5.2 Geo-Sharding and Distributed Spatial Indexing ....................................................................................... 23
  5.3 Dimensional Limitations............................................................................................................................. 27
6 Benchmarking Testbed and Experiments.................................................................................................. 28
  6.1 Test Data Characteristics........................................................................................................................... 30
  6.2 Physical Infrastructure Characteristics........................................................................................................ 31
  6.3 Access Restrictions and Security Implications.......................................................................................... 32
7 Conclusion and Recommendations ........................................................................................................... 33
  7.1 MongoDB High Scale Prototype – No Cost Contract Recommendation .................................................. 33
  7.2 MongoDB High Scale Prototype – CRADA Recommendation ................................................................. 34
  7.3 Conclusion.................................................................................................................................................. 34
A Scalable Indexing Technique and Implementation for Big Spatial Data

List of Figures

Figure 1: iSpatial Core User Interface .......................................................................................... 6
Figure 2: iSpatial Interface used for 3D User Defined Operation Picture Widget.................... 8
Figure 3: Ubiquity Web Dashboard ............................................................................................... 9
Figure 4: MongoDB large-scale deployment service architecture (source: 10gen) ................. 12
Figure 5: MongoDB software implementation and file structure ............................................. 15
Figure 6: MongoDB logical data structure ................................................................................... 17
Figure 7: MongoDB's “off the shelf” point in polygon algorithm uses simple raycasting ................................................................. 19
Figure 8: MongoDB's original spatial $within search operator required a vertex lie in the search polygon .................................................................................................................. 19
Figure 9: Logical view of MongoDB's data structure with R-Tree spatial index enhancements ........................................................................................................................................ 21
Figure 10: Minimum bounding rectangle representing KeyNodes for the R-Tree indexer .................................................................................................................................................. 22
Figure 11: R-Tree data structure and graphical example with points and rectangular polygons ............................................................................................................................................. 22
Figure 12: MongoDB's sharding implementation attempts to keep each shard as balanced as possible (source: 10gen) .......................................................................................... 24
Figure 13: Simple Geo-sharding example where shards are evenly distributed over a space – reality will not be so perfect ............................................................................................ 26
Figure 14: Example R*-Tree split across 3 shards ...................................................................... 27
Figure 15: R*-Tree CPU and Page access performance compared to X-Tree as a function of data dimensionality (source: Kriegel 1996) ................................................................. 28
Figure 16: Example MongoDB statistics on a small deployment ............................................... 29
Figure 17: Example infrastructure for distributed indexing across multiple data centers 31
Figure 18: Example document with 'user' tag applied to provide cell-based data access. 33
1 Executive Summary

TST is a small business that combines prior service operations and intelligence personnel with highly technical engineers, scientists, and developers. Our core focus is on solving mobile, geospatial, and cyber challenges in the defense community. We have dialogued with many offices of NGA over the last five years and currently provide a small level of development support to this organization. Based on the NGA National Expeditionary Architecture Cloud Elements for the Global Enterprise, we believe some of the work we have done has specific relevance at this time.

2 Introduction

2.1 TST Business Approach

TST provides Information Technology (IT) services to the US Government but also has a significant customer base in the commercial space. The primary commercial sectors are Oil, Gas, Energy, Utilities, Marketing, and Sports. TST focuses on developing products around emerging technologies that support a common denominator among a breadth of customers. This approach was specifically designed in order to aggressively pursue emerging technologies and apply them to our products in support of customers in Commercial and Government sectors.

TST works to adopt open source technologies as often as possible when compiling software products and solutions for customers. Many times, the open source solutions that exist are missing key components that would enable them to fully support the products we develop or our customer’s specific needs. TST maintains an open source test bed where our development team contributes source code directly to open source projects that are leveraged within our products. This unique approach provides us a degree of objectivity when addressing various challenges. It also adds modularity, openness, and a greater breadth of solutions when solving customer’s technology challenges.

Over 4 years of working with commercial and Government customers, TST has immersed themselves in managing common denominator requirements around mobile and geospatial solutions. The underlying data, data service, and web services that support our products are the core components we are able to invest in that bring commercially funded updates to our Government customer base. As a small company, the requirements that both communities have are evaluated for development. The hardest challenges that are not solved by others in the IT space are the ones we generally work to solve. Our goal is to significantly increase our software sales in the commercial space to fund
extensive research and further development in both our products and open source technology.

2.2 Current Customer Examples

TST supports the following customer’s enterprise systems with a focus on their geospatial, cloud, and mobile technology solutions:

- US Army G2 - uses iSpatial to track soldiers’ locations via smart phones and to organize data collected by soldiers in the field which is then displayed over Google Earth for use by other soldiers and decision makers.
- US Southern Command - uses iSpatial to share information with other intelligence agencies on classified networks, as well as share unclassified data with partner nations and non-government organizations in Latin America.
- The Department of State Bureau of Diplomatic Security - uses iSpatial to track security teams in high threat areas. It is also used to monitor and coordinate US military movement in countries where they operate.
- US Army G2 – uses Ubiquity to connect into cloud computing systems that allow Soldiers to collect data while in the field and send that data back to their base for immediate analysis and collaboration.
- NSA – TST provides development support on classified CNE programs.
- DARPA – used Ubiquity as prototype to provide non-government organizations in Afghanistan a common tool for reporting on current events.

3 Current Capabilities and Future Plans

iSpatial and Ubiquity represent mature user facing solutions that are leveraged to fully test out the theories developed in our Applied Sciences Group (ASG). A brief overview of these capabilities is listed below. They would be just one example of the test interfaces used to support a potential prototype effort.

3.1 iSpatial – Geo Visualization Framework

iSpatial is a geo-visualization framework that enables users to easily build web-based solutions to author, manage, share, and collaborate within their secure enterprise over the browser-based 3D Google Earth or 2D Google maps. iSpatial was designed to give an organization a complete framework to rapidly take new and existing data no matter what system it was in and represent it via a web interface geospatially. iSpatial deals with static legacy data as well as real-time operational data such as current locations of vehicles / cell phones, future planned activities, simulation data, and tour/fly-through data.
A Scalable Indexing Technique and Implementation for Big Spatial Data

iSpatial is comprised of an enterprise geo store, a geo index, a collection of geo services (REST/JSON), web visualization libraries (Javascript), and core user interfaces. It has industry specific modules that can plug-in and augment the core capability in order to rapidly achieve a full operational capability. iSpatial is tightly coupled with the Google Earth Enterprise suite of capabilities and natively connects to the Google Earth Browser plug-in and the Google Maps Browser interface. Its primary use is to geo-enable existing or new web applications over Google Earth or Maps in a web environment.

A key strength of iSpatial is its scalability, which allows thousands of users to simultaneously access it via the web. It is designed to run in a cloud environment with a backend that processes millions of geo objects, displaying tens of thousands of objects at one time in a browser, conducting near-real-time refresh of the data being ingested and displayed, and supporting thousands of concurrent users.

iSpatial was made available for Government use as an alpha release for the first time in 2008 for the Bureau of Diplomatic Security at the US Department of State. It was rapidly adopted as a core component of their Next Generation Blue Force Tracking
system. The software gained further operational Government use when it was selected as the framework for Operation Unified Response (OUR) as the US and other nations responded to the 2010 earthquake in Haiti. While designed for commercial use the Federal Government’s strong interest in the framework led to organizations like the Defense Advanced Research Programs Agency (DARPA), Army G2, and National Intelligence Agencies leveraging it for us in their enterprise systems. Many customers used an enterprise licensing approach, with the largest to date occurring when the US Army Military Intelligence community procured an enterprise license to leverage the framework for their core web based geo visualization. An increased effort has been made since that time to expose all iSpatial functionality from the commercial sector for US Government use.

The concept of Geo Literacy is taking hold throughout commercial and federal users. Enterprise Information Management systems have historically shown data in a way that lacked a geospatial view or provided a 2-Dimensional view that was not dynamically connected to a data store. The field of Geospatial Information Systems (GIS) was specific to a group of subject matter experts but the geo tools were not used in the day-to-day operations of an organization writ large. The advent of web tools like Google Maps, Bing Maps, Yahoo Maps, and Google Earth have dramatically changed the landscape and users are well versed in managing their data in 2D or 3D geospatial format.

iSpatial was designed as a geospatial framework to rapidly deliver geo functionality to any user with little to no training requirement. When an organization looks at their requirements to visualize and interact with geospatial data there are a standard set of “common denominator” components that go into the solution they build. Those components were identified and developed in iSpatial. The software has been refreshed with updated data architecture elements, updated interaction with web and mobile interfaces, and with updated geo services. As new technology emerges in managing geospatial data, iSpatial evolves its internal architecture to leverage those advances. The latest release is the 2.0 version that was made publicly available in February of 2012.

As iSpatial continuously improves as a geospatial framework it is continually enhanced by its breadth of active users. This establishes an economy of scale around each feature within the software and the development team streamlines each component based off of user feedback. If an organization was to build a geospatial component of their existing system and bear the full burden of continued research and development it significantly increases their total cost of ownership. The modular approach that iSpatial customers take is to use Google Earth or Maps, use iSpatial, and perform small
customizations to the final web interface. Because the software is web based it makes after-market updates very simple.

In contrast to being developed as an end-state capability that is difficult to customize or requires specific knowledge of the product, iSpatial is an open software framework. This means the Application Programming Interfaces (API), software libraries, and customized modules are readily accessible by any developer. The iSpatial APIs enable an organization to springboard off of the existing framework and develop specific features needed for virtually any use case. Organizations have been able to go to production with iSpatial in as little as 2-3 weeks by leveraging the existing framework.

Figure 2: iSpatial Interface used for 3D User Defined Operation Picture Widget

iSpatial consists of a front end browser based interface developed in Javascript and ExtJS, a collection of REST services that connect the user interface to the back end, and a PostgreSQL/PostGIS and MongoDB backend. It has Oracle and Google Fusion Tables extensions that can be integrated as needed, depending on the deployment model.

3.2 Ubiquity – Mobile Application Framework

Ubiquity is a mobile application framework that enables users to dynamically provision native mobile application functionality via a web-based user interface using abstract widget components and REST/JSON services. Ubiquity empowers users and organizations with the ability to create new mobile applications on the fly and gives them
the flexibility to *update or modify, in near-real-time*, existing applications and functionality without requiring native application updates on the device.

Ubiquity was designed for everyday users to leverage an existing web-based user interface to compile various capabilities and mash up a collection of widgets for use on multiple mobile smart phone devices (presently Android 2.3+ and iOS 4+ capable devices). The increased interest by the Federal Government led to development of a series of additional Application Programming Interfaces (API) to provide enhanced and/or new capabilities. Ubiquity was originally designed and developed for commercial use with Android and iPhone device, but has rapidly been adopted within the Government sector. Figure 3 shows the web interface that users access for dragging and dropping the widget functionality they want to see in their mobile application(s).
Ubiquity was made available for Government use as an alpha release for the first time in Operation Unified Response (OUR) as the US and other nations responded to the 2010 earthquake in Haiti. While designed for commercial use, the Federal Government maintained an interest in the framework and organizations like the Defense Advanced Research Projects Agency (DARPA) leveraged some of its capabilities in various mobile studies and prototypes. A key investment was made in Ubiquity when the US Army Military Intelligence community procured an enterprise license to leverage the framework to build and manage all mobile applications.

With the Ubiquity platform as its backbone, Windshear securely provides the warfighter with analytical capabilities, biometrics, and geospatially-based intelligence reports pushed from the top operator—all in one application. Through utilizing the modular nature of Ubiquity’s widgets, the warfighter can easily provision based on their immediate needs. An increased effort has been made since that time to expose all Ubiquity functionality from the commercial sector for US Government use.

The U.S. Southern Command (SOUTHCOM) developed a mobile application, based on the Ubiquity platform, which allows command personnel to report their locations during a recall. In the event of a destructive weather event or other disaster, command members could update their status and location through their smartphone application, and in turn their spot reports would be viewable on the Recall instance of the 3D UDOP.

### 3.3 Future Plans

TST plans to continue developing core products and work towards highly scalable versions that can be dynamically provisioned for use by any user, whether commercial or Government. Additional internal research and development will continually expand our capabilities around the area of cloud, geospatial, mobile, and cyber (CNA/CND) solutions.

TST is investigating various open source Hacktivist OS Distributions. Current research is focused on Backtrack 5 (BT5). In short, BT5 is an Ubuntu flavor operating system with built in hacker tools for cyber offensive and defensive activities. The end goal is to study BT5 and identify how specific components might support future TST cyber products from both cyber offensive and defensive perspectives. Additional research is underway to port components of this OS to Android to develop special operations mobile utilities.

TST’s experience with one of the most popular open source packages for spatial statistics and econometrics tailored for Geo-Analytics is a tool called R. It contains a significant amount of pre-developed algorithms that are on par (and arguably more
A Scalable Indexing Technique and Implementation for Big Spatial Data

effective) with tools like ESRI’s ArcObject toolbox. It also contains one of the first parallel computing extensions for spatial analytics. PostGIS has an API interface to R already; Hadoop Map/Reduce currently provides a distributed processing framework but requires developers to write the Map/Reduce algorithms. Discovery is underway to develop R and/or Arc Toolbox support for the iSpatial indexing and Map/Reduce framework. This enables us to leverage the wealth of existing algorithms along with the Map/Reduce processing framework to bring complex spatial processing directly to the data.

Additional activity is planned around continuing on the MongoDB development for a selective replication strategy based on location. This work will bring MongoDB server components to Android. This will also support data replication around MongoDB that is based on user location and the type of data they are interested in.

Introducing mobile devices to thousands of users compounds the geospatial data management challenge that organizations face. Tracking locations of many different phones, having them submit data of a highly spatial nature, and interacting with many moving objects in a spatial environment led to the need for enhanced research in the high scale geo data management arena. Additionally, conducting advanced geospatial operations with data at scale brought challenges that current software and database solutions have not effectively solved. The following sections 4-6 detail TST activity and development in solving some of these challenges in order to make Ubiquity and iSpatial function effectively at scale.

4 MongoDB Architecture, Organization, and Data Structures

The MongoDB architecture is comprised of the following four software components:

1. mongod – the primary MongoDB server process/service that provides the database storage and indexing logic along with simple configuration capabilities for managing a single instance or a cluster.

2. mongos – the routing and coordination process/service that abstracts individual cluster components from a client application. Mongos makes the distributed components appear as a single system.

3. gridfs – specification for storing extremely large files (e.g., multimedia, archives) as mongo documents so they can be accessed and indexed by the database system.

4. mongo – administration client application for managing a large or small MongoDB deployment.
Using these four architecture components the MongoDB system provides **elastic scalability** – the ability to commission or decommission resource capacity on the fly – for supporting rapidly changing system needs and requirements. This feature can be achieved either by manually provisioning new hardware within a physical data center, or automatically provisioning virtual hardware within a hypervisor or distributed virtual environment. The MongoDB services are deployed within the hardware environment and dynamically configured based on data access and scale demands. Figure 4 presents the logical layout for a Large-scale MongoDB deployment.

Horizontal scaling in MongoDB is achieved through **sharding** capabilities (indicated in Figure 4 by shardX) provided by the mongos and mongod services. The data-partitioning scheme for the shard configuration can be configured statically by specifying a key range for each shard or dynamically (auto-sharding) by leaving the partitioning logic to the mongos process. Since the mongos process(es) will automatically handle both load balancing and data distribution across shards, auto-sharding is recommended for most configurations. However, it is important to note that some system designs might benefit from static partitioning. Geo-sharding, for example, can specify key ranges based on geographic locations. Using static partitioning schemes in a geo-sharded environment, each shard can be dedicated to a specific geospatial region and contain only those data whose geo-positioning falls within the shard’s designated region.

To achieve high data availability and safeguard against inevitable hardware failures MongoDB uses **replica sets** – an asynchronous master/slave configuration that provides
A Scalable Indexing Technique and Implementation for Big Spatial Data

automatic-recovery to the member nodes. Similar to transactions in RDBMS technology, but more flexible, replica sets provide a form of dynamic data consistency, isolation, and durability. A replica set consists of two or more data nodes that are copies of each other. One node is designated as the primary (master) the other as the secondary (slave). Using a capability known as journaling, the mongod process replicates data from the primary node across the secondary nodes. To achieve high-availability the mongod processes automatically designate a secondary as the new primary in the event a primary node is inaccessible (e.g., node failure). Data consistency requirements can be dynamically specified in the application layer at run-time using the keyword slaveOk. This feature allows an application to read data from any secondary node (replica server) understanding that the secondary node’s data might be inconsistent with the primary. MongoDB specifies slaveOk=false by default to ensure high data consistency requirements (similar to RDBMS MVCC requirements found in Postgres but optimized for large datasets).

4.1 Software Implementation Overview

MongoDB maps on-disk data files directly to in-memory data structures where the access logic is implemented using pointer arithmetic features of C++. This design choice enables the system to manage memory more CPU/time efficiently than other database systems implemented in Java or other derivative languages. Cost-effective memory management is an important feature of MongoDB and must be carefully considered by developers when implementing feature enhancements, system extensions, or bug fixes. Alternative database system implementations, however, transfer this memory burden to native garbage collectors so developers can focus more on solving problems than worrying about platform details. Unfortunately, this design choice reduces development time at the cost of implementation performance. While the performance trade is negligible for average sized systems a database system designed for unrestricted data scale, while retaining beneficial properties such as flexibility and speed, will experience an exponential decrease in read/write performance as memory usage and garbage collection invocations grow. For moderately high data requirements (e.g. terabytes) the Java garbage collection system is, in theory, as CPU/time efficient as unmanaged memory in C++. However, the garbage collection system requires 5-10x more memory to achieve the same performance – and the problem is exacerbated for systems reaching extremely high “big-data” capacity (e.g., Zettabytes - 10^21 bytes). These performance/development trades are extremely important to consider as there is often a willingness to regress on system performance as hardware becomes faster and cheaper. MongoDB uses faster/cheaper hardware for the benefit of faster read/write performance over larger amounts of data.
Due to the critical importance of efficient memory management the MongoDB implementation is extremely careful with stack and heap allocations using smart-pointers, auto-pointers, and shared-pointers where applicable. To make this implementation burden more manageable the codebase heavily leverages the performance benefits provided by the Boost companion library. Backed by the C++ Standards Committee, Boost is an open-source set of standard C++ companion libraries designed to provide optimized implementation capabilities for large-scale projects. To minimize the complexity of system deployment Boost is the only library MongoDB requires with the rest of the data structures and functionality explicitly developed and provided in the MongoDB codebase.

Figure 5 provides an overview of the memory-mapped functionality implemented in the MongoDB core. At minimum there are 3 on-disk memory sequential files provided in the database /data directory (may reside on a distributed file system such as GridFS or HDFS). These 3 files – test.ns, test.0, and test.1 in our example – provide all namespace, indexing, document, and collection information for the local member node as well as details about the cluster configuration (e.g., for sharding and replication). The *.ns file provides DiskLocation pointer references to the first and last data extents that are contained in the *.0, and *.1 data files, respectively. Document data (BSON documents) for each collection are contained within the *.# files. The number of *.# files depends on the amount of data stored on the member node and exponentially increase in size with each increasing numbered extension. For example, the first data file (*.0), is allocated with $2^{26}$ bytes and the second file (*.1) is allocated with $2^{27}$ bytes. As data requirements increase this process continues to a maximum of $2^{34}$ bytes (2GB) – to be increased in upcoming versions. Top-level index data (e.g., key pattern, specification, and root node) is also contained in the *.ns file with index details stored in each extent.
Existing indexing implementation stores index data in memory using a hybrid B-Tree data structure (see section on Data Structures). Indexes can be built in either offline foreground or online background mode. When indexes are built in foreground mode the entire database will be locked such that no reads or writes will be allowed. This is designed for situations when the database contains many data records where the index has not already been built. Alternatively an index can be built in the background where the system will still respond to reads and writes while the index is being built. In background mode, however, the performance of the database will decrease as resources are allocated for compiling and writing index details. MongoDB supports a wide range of indexing configurations and capabilities (e.g., embedded files, compound keys, document keys) most of which are too detailed to cover within the scope of this paper. The important implementation detail to note, however, is that MongoDB orders keys based on the canonical or numeric comparison of the field value (also true for embedded document keys). In this manner MongoDB’s indexing capabilities are limited to single dimension data only – a limitation for spatial data discussed further in section 4.3.
4.2 Data Structures

The following list describes the primary data structures implemented in the MongoDB codebase:

1. BSONObj – in memory representation of a binary object stored on disk.
2. DiskLoc – represents a disk location ID and offset location within a memory mapped file. DiskLoc objects ultimately point to the underlying BSON objects stored within the document files (e.g., *.0, *.1, *.2).
3. NamespaceDetails – represents an entry in the database namespace file (e.g., *.ns) and provides all of the information necessary for accessing the underlying documents and building the index in memory.
4. BucketBasics – helper class used to provide access and control functionality for the underlying index data structure.
5. BtreeBucket – represents a single node in the index structure.
7. IndexDetails – provides all of the information for building the index in memory including the root pointer and key format.
8. IndexPlugin – a base class for developing new index capabilities.
9. IndexType – singleton object that represents a cached in-memory index for quick access.
10. Cursor – a base class used for iterating over result objects found during query.
11. KeyNode – a type class that contains key details and functionality for computing the key.

Each of these data structures can be extended or modified for providing customized functionality, at the data level, based on capability needs or nuances of the specific problem space.

The doubly-linked-list is the principal data structure used for creating and accessing in-memory BSON objects, disk locations, and index records and ties together each object, described above, through the use of memory managed “smart pointers” (leveraged by the BOOST companion library). Through this useful data structure each extent, record, and index object is easily accessible using simple pointer arithmetic. This design enables a compact memory footprint eliminating the need to worry about unnecessary memory allocation or force garbage collection.

The data index is implemented using a hybrid B-Tree data structure where each node of the tree, called a “bucket”, maintains a linked list of KeyNode objects that reference back to the underlying BSONObjects. In this hybrid approach the underlying B-Tree more closely resembles a binary tree such that each bucket only contains left and right children – represented again using a doubly linked list. Each bucket contains $n$ keys,
which is index specific, where \( n \) is determined by the size of the keys plus the size of the disk location divided by the size of the bucket. In this manner each index will have a different number of keys available in each bucket. This allows the index data structure to be flexible based on the requirements of each index specification (e.g., order, capped). Figure 6 provides a logical representation of the relationships between each of the data structures listed above.

![Figure 6: MongoDB logical data structure](image)

Document updates are atomic operations that occur in place without the use of transactions – as found in traditional RDBMS software. This allows updates to occur in near-real time without the need for global write-locks, which carry the risk of deadlocking and speed reduction. If document modification increases the size of a record beyond the original allocated space the entire record is moved to a new location big enough to contain the new document. The original document space is then marked as deleted and added to the list of free and deleted documents that is referenced within the namespace details. While this could potentially lead to fragmentation issues, a dynamic compaction algorithm efficiently organizes the documents to reduce or eliminate this issue.
4.3 Spatial Implementation and Limitations

Current open-source MongoDB versions only support two types of geospatial indexing capabilities: 2D and geo-haystack. The former is by far the most widely used as it enables a wide range of spatial query functionality (e.g., bounding box, nearest neighbor). The typical spatial applications for MongoDB consists of simple point queries as polygonal indexing and searching is highly limited due to the design and specialized implementation of its hybrid B-Tree indexer. Since MongoDB’s indexing capabilities were originally designed to support single dimension indexes only (by far the most popular use of database technology) spatial support heavily relies on the use of dimensionality reduction to transform multiple dimension data into a single dimension index. For implementation simplicity the dimensionality reduction technique used in MongoDB is the Z curve. This technique is implemented by converting longitude-latitude pairs into an interleaved unsigned 64-bit geohash. In spirit of the highly flexible optimized performance design of MongoDB the 2d spatial index specification permits the use of a bit size parameter for tuning the spatial resolution of the index. By default this parameter is set to 26-bits, representing ~1 foot on the earth’s surface (note: MongoDB assumes a perfectly spherical earth so this resolution actually varies based on the location on the earth), but can be specified up to a maximum of 32-bits. This, among other reasons, is why MongoDB is designed to run on 64-bit architecture only.

As of MongoDB development version 1.9 polygon support is made available in the form of multi-location document indexing. This capability is actually implemented by indexing multiple locations that reference back to the same BSON document. If not careful this capability can consume a significant amount of memory resources for a single document and is often abused for indexing complex geometry (e.g., circles, ellipses). In this situation each point along the perimeter of the circle is indexed separately. While each individual point consumes little memory the aggregate of each point for every geometry object can quickly consume more memory than documents on disk.

Spatial queries in MongoDB’s 2D indexer are designed specifically for individual point queries (e.g., nearest neighbor) and optimized for speed. Nearest neighbor queries are implemented using the great-circle distance formula (e.g., final all points within 30 miles of this location), while bounding queries are implemented using the simple ray-casting algorithm found in the popular CLR “Introduction to Algorithms” book. Figure 7 illustrates the basic point-in-polygon algorithm used for bounding queries.
Figure 7: MongoDB’s “off the shelf” point in polygon algorithm uses simple raycasting

Multi-location document indexing uses the same nearest neighbor and bounding query implementations for locating “polygon” documents that fall within a specific search distance or within a search polygon. Unfortunately these algorithms were never originally designed to support polygon representations and fail when a target vertex does not fall within the designated search polygon. In other words, polygonal clipping is not implemented in MongoDB. Figure 8 illustrates a situation where the target polygon will fail with a bounding query.

Figure 8: MongoDB's original spatial $within search operator required a vertex lie in the search polygon
In addition to basic 2d indexing MongoDB supports geo-haystack indexing. This indexing technique is highly discouraged for applications requiring complex spatial queries or queries over a large distance. In short, geo-haystacking is a type of **bucket indexing** where multiple locations containing a common attribute (e.g., type) and located within a certain distance of each other (e.g., store all locations with name ‘foo’ that are within 1 unit of lat/lon of each other) are stored within the same bucket. Due to the size limitations of the bucket, this indexing technique is useful for small area queries but has difficulty scaling over large quantities of spatial data.

### 5 MongoDB Spatial Improvements

Geospatial indexing in MongoDB was developed to provide basic point-based indexing for enabling simple spatial query capabilities in support of companies developing simple location-based services and location-aware applications (e.g., Foursquare). While the indexer was never intended for deep Geospatial Information System (GIS) applications and analytics the scalable document-based framework provides a rich foundation for building a spatial processing framework designed and optimized to support **big spatial data**. To overcome the limitations of the existing spatial index (e.g., dimensionality reduction, 2d only) an entirely new indexing structure and design is required. The R-Tree indexing technique, originally proposed in 1984 by Antonin Guttman, is one of the most popular spatial access methods (SAM) for indexing and querying complex spatial data. It is the foundational data structure for popular relational database systems such as Oracle’s spatial extension and PostgreSQL’s GIS extension (PostGIS). It provides the ability to index multi-dimension data without changing the basic structure and theory of multi-way height balanced trees – the computer science foundation for most external search tree structures – that provides the basis for MongoDB’s B-Tree indexer. The fundamental difference between B-Tree and R-Tree data structures are found in the key specification, and the node split, merge, and balance algorithms while the rest of the structure remains relatively the same. This makes the R-Tree implementation a natural fit for the existing MongoDB framework.
A Scalable Indexing Technique and Implementation for Big Spatial Data

Figure 9 presents a logical view of the existing MongoDB data framework with the R-Tree spatial data structures and index enhancements. The design choice is to minimize the amount of modifications to the existing MongoDB core codebase while providing an enhanced spatial index support framework that leverages as much of the existing data structures and design as possible.

5.1 R-Tree Indexing Algorithms and Implementation

A significant number of R-Tree implementations and variants can be found in the academic and professional literature. While each implementation carries a different label (e.g., R*-Tree, R+-Tree, Hilbert R-Tree) the underlying structure and theory remains the same. The primary difference between each R-Tree variant can be found in the implementation details of the node requirements, key specification, and the split, merge, and balance algorithms. Our R-Tree implementation generally follows the techniques outlined for a standard R*-Tree (Beckmann and Kriegel, 1990) with a few modifications and optimizations to the insert and split algorithms. Like the traditional R-Tree, the KeyNode for complex geometry documents in the R*-Tree is represented as a minimum bounding rectangle (MBR) that encloses all dimensions of the object and, like the traditional R-Tree, the inner nodes themselves contain a MBR that encloses all of the node’s children. Figure 10 illustrates the MBR for geometry objects in 2 dimensions and Figure 11 provides an illustration for the R-Tree data structure with a graphical example of geometry documents. A MBR in the 3rd dimension is represented as a cube and hyperspatial-cubes are used for the 4th dimension and beyond.
The node split, and merge algorithms implemented for insert and delete operations of a standard R-Tree, respectively, is based on a heuristic optimization criterion that minimizes just the area of each enclosing rectangle within the inner nodes. Beckmann and Kriegel have shown that this criterion is often taken for granted and not always the best possible criterion for determining the best possible insert path. Our R*-Tree implementation adopts a four degrees of freedom optimization that depends on the following criterion:

1. Area reduction – minimize the area covered by the bounding rectangle but not covered by the enclosing rectangles (e.g., the dead space).
2. Overlap reduction – minimize the overlap between the rectangles of the inner nodes.
3. Margin reduction – strive for each inner rectangle to be as close to quadratic as possible (e.g., as close to square as possible since oblong rectangles reduce the probability of additional inserts).

To optimize query performance over large volumes of data the R*-Tree implementation adds a fourth criteria that strives to maximally optimize the storage utilization of the inner rectangles. In other words, we try to fit as many children in the enclosing MBRs as possible. This optimization approach tends to reduce query cost since it keeps the height of the tree as small as possible. It is important to note the high correlation between these four criteria. For example, minimizing the area and overlap regions tend to negatively impact storage utilization since it tends to reduce the number of children that can be stored in the inner node. Similarly, area and overlap minimization may negatively impact the shape of the covering MBR such that they are no longer quadratic. This implementation strives to balance these optimization criteria as much as possible through the selection of static threshold parameters. These parameters (proven by Beckmann and Kriegel’s spatial testbed) are selected such that read/write performance is optimal for extremely large data sets and sub-optimal for smaller data sets (e.g., on par with RDBMS performance). While these threshold parameters can be tuned to the characteristics of the expected data space this implementation has no desire to add supplementary parameters to an already intuitive MongoDB configuration framework. While future considerations may include the ability to dynamically change threshold parameters such that indexing performance is optimized as the data set grows initial results suggest a dynamic optimization approach will require a massive tree rebalancing effort that may impact real-time performance.

5.2 Geo-Sharding and Distributed Spatial Indexing

To horizontally scale enormous amounts of data writes MongoDB relies heavily on its auto-sharding capabilities (see Section 4 for a brief explanation of sharding). When configuring the sharded cluster a partition key, similar to an index key pattern, is passed to the mongos process to define one – or many – field(s) upon which the data will be distributed. The mongos process uses this key pattern to divide the collection’s BSON documents into a contiguous range of data, known as chunks, and balances those chunks across the multiple shards. To take advantage of MongoDB’s auto-failover and consistency capabilities each shard is typically configured as a replica set. The mongod config server process(es) stores shard details in its own database as a collection, minKey, maxKey, and shardLocation document. The chunk details for each collection are represented as a \{collection, minKey, maxKey\} tuple and stored in the namespace
files of the mongod process located on each shard server. When a data write, upsert, or update occurs (e.g., BSON document insert or modification) the mongos process uses the partition range of each shard (obtained from the config server) to identify the mongod shard process to handle the command. The command and document are then passed to the mongod shard process whose data partition range bounds the key of the target document. The mongod process uses the namespace details object (stored in the .ns file) to identify the chunk for which the document is inserted. When a chunk reaches its maximum capacity (tracked by chunk statistics in the mongos process) the mongos process sends a command to the mongod server to invoke a chunk split – similar to a node split in multi-way height balanced tree logic – that divides the growing chunk into 2 smaller chunks. To ensure an even distribution of data across shards the mongos process contains balancing logic that statistically tracks the number of chunks on each shard server. When an unbalanced condition is detected the mongos process sends a command to the primary server of the overloaded shard to invoke a migration process. This process occurs online and could potentially take a long time depending on the amount of data on the shard server. During this time read/write requests are still serviced though performance degradation is expected. To overcome this performance impact replica servers are extensively used to service those requests with lenient consistency requirements (e.g., slaveOk=true). Figure 12 provides a graphical illustration of MongoDB’s sharding logic.

![Figure 12: MongoDB's sharding implementation attempts to keep each shard as balanced as possible](source: 10gen)
A Scalable Indexing Technique and Implementation for Big Spatial Data

Careful selection of the partition key is critical for achieving optimized sharding performance. As illustrated in Figure 12 the full range of the partition key space is divided into chunks and evenly distributed across the shards. Unfortunately this requires a single dimension key space that makes multi-dimension data distribution difficult across shards.

While the existing geo-sharding approach works as a result of the 2d dimensional reduction process of the geohash, it does not apply to R*-Tree based index keys. Therefore, enhancements to the existing shard logic are required for partitioning across multiple servers. The R*-Tree geo-shard functionality are divided into two levels of enhancements – with ongoing research to quantify the best geo-shard approach. The first enhancement leverages relatively recent research in multi-dimension Hilbert curves for R-Tree data structures proposed in 2009 by Haverkort and Walderveen. With this technique the Hilbert-order of the enclosing MBRs is used to guide chunk insertion within the partitioned key space. In other words, the space-filling curve is used on the bounding rectangles (quads in 2d, octants in 3d, hexadecants in 4d) to determine a scanning order for insertion within the multi-dimension key space. This technique is accomplished using a binary representation for the order of the enclosing hyper cubes. For 2 dimensions this order is the simple geohash, for 3 dimensions the order is selected similarly to Oracle spatial’s Helical-Hyperspatial Code (HHCODE) approach, and for 4 dimensions a rotation of the Alber-Niedermeier (H4cdANR) scanning order, as proposed by Haverkort and Walderveen, is used which provides significant improvement over the traditionally implemented Peano curves for big spatial data. The second enhancement leverages the SD-rTree data structure originally proposed in 2007 by Litwin, Mouza, and Rigaux. The SD-rTree structure is a scalable distributed data structure (SDDS) that adds shard distribution logic to the selected R-Tree implementation of choice (e.g., R*, R+, Hilbert). In this implementation each shard server (mongod process) contains its own R*-Tree index for each chunk where the root represents a different MBR within the spatial coordinate system. The config server maintains the root image for the distributed tree that the mongos server process uses to determine which shard contains (on modification) or will contain (on insert) the target BSON document. If this image is outdated (due to a working split invocation) the config server is automatically updated using feedback from the mongos process(es). Like the single dimension case, when a chunk becomes overfull as a result of massive inserts or modifications, the mongos process invokes a split operation on both the chunk and tree representation of the chunks partition key. This split operation is the same operation carried out on an overfilled tree node. The tree is divided into two equal trees for each chunk and one chunk is moved to an underworked shard. This implementation leverages the shard logic for the existing
A Scalable Indexing Technique and Implementation for Big Spatial Data

MongoDB codebase while adding minor enhancements optimized for the R*-Tree data structure.

Figure 13: Simple Geo-sharding example where shards are evenly distributed over a space – reality will not be so perfect

Figure 13 provides an oversimplified illustration for the geo-sharding logic provided with the multi-dimension spatial enhancements. Figure 14 provides a graphical illustration of the R*-Tree data structures stored on each shard server (known as GeoShards). The benefit of the SD-r*Tree implementation is that the balance logic tends to migrate chunks that are within a nearby geographic location. Initial performance metrics suggest a worst-case complexity of $2 \log N$ for insert and balance operations over $N$ servers and $\log N$ for polygonal and point queries. This implementation, however, is still in research and development phase and not expected to be ready for prime time until later this year at the earliest (2012 at time of writing).
5.3 Dimensional Limitations

With the growing need for 3 dimensional indexers to support altitude and 4 dimensional indexers to support time, the R*-Tree structure has been implemented and integrated as a horizontally scalable multi-dimension indexer in MongoDB. Understanding the structure’s limitations, however, is extremely important for deciding which problem space the indexer is best suited. This is an ever-growing issue as integrators often assume a spatial structure will work for any spatial problem or try to develop a solution far removed from the source of the problem (e.g., within the application layer or by throwing more hardware at the issue). Just as single dimension indexers (e.g., B-Tree, K-d Tree) are inefficient with hashed 2 dimension data, multi-dimension indexers are inefficiently applied to hyper dimensions. But what is the dimensional limitation? While this question is still very much under considerable research a significant amount of work provides initial guidance to the limitations of the presented SD-r*Tree multi-dimension indexer. In a 1996 research publication, Berchtold, Keim, and Kriegel – researchers of the original R*-Tree variant – present a detailed evaluation of the important characteristics and limitations of the R*-Tree data structure. It is noted that inner-node overlaps rapidly increase as the dimensionality of the data grows. The increase in overlap negatively impacts both query and insert performance since more nodes need to be searched when locating a target document. Figure 15 provides a graphical illustration of the page access and CPU performance for the R*-Tree as data dimensionality increases. Also provided on the graph is the performance of an alternative index structure called the X-Tree that is optimized for higher dimensional data. This chart reveals negligible performance improvement for data
higher than 4 dimensions. As a result, the implemented SD-r*Tree data structure for MongoDB is limited to a minimum of 2 dimensions and a maximum of 4 dimensions. The X-Tree implementation is under development for supporting beyond 4 dimensions and is projected to be available as early as 2013 (at the time of writing). This, of course, assumes the need for higher dimension indexing capabilities to support applications requiring vector based processing. For example, a higher dimension indexer is optimally suited for analyzing spatio-temporal hyper-spectral data of the vector form \([x, y, z, t, b, f]\) where \(x = \text{longitude}\), \(y = \text{latitude}\), \(z = \text{altitude}\), \(t = \text{time}\), \(b = \text{spectral band}\), \(f = \text{feature vector}\).

![Figure 15: R*-Tree CPU and Page access performance compared to X-Tree as a function of data dimensionality (source: Kriegel 1996)](image)

6 Benchmarking Testbed and Experiments

New distributed storage and indexing techniques (such as those presented in this whitepaper) designed to support a wide range of heterogeneous multi-/hyper-dimension spatial data drives the need for relevant test data and environments for evaluating system performance under varying conditions. Legacy spatial systems, built on first generation relational databases, struggled to support user’s spatial needs and requests when data outgrew the limitations of the system design (e.g., scale, dimensional limitations). This resulted in high latency for even the simplest queries such as, “find location of object X,” “find objects within distance Y of location X,” and “find 10 nearest objects to location Y.” The spatial indexing extensions to the MongoDB framework are designed to support more advanced queries such as, “find all objects X within three-dimesional space \([\text{lat, lon, alt}]\) over time span T,” or “find all pixels X within two-dimensional space \([\text{lat, lon}]\) with SWIR signature range S and time period T.” To diligently benchmark the performance of the indexer for characterizing the use-case limitations of the proposed system requires a significant benchmarking testbed environment with access to large
volumes of spatially referenced data. Access to such an environment enables the ability to execute a number of statistical experiments and performance analyses leveraging benchmarking tools implemented at the indexing and storage layer.

Figure 16: Example MongoDB statistics on a small deployment

Figure 16 illustrates top-level health status and statistics for a small MongoDB deployment (3 shard configuration with auto-failover). The statistics tool (mongostat) provides real-time updates about each deployed node as clients (users and services) interact with the system. The following columns are of particular importance when assessing the performance of the distributed spatial index. (Note: these descriptions are provided in the standard MongoDB documentation and apply to the R-Tree spatial extension enhancements).

- **insert** – The number of document objects (BSON) inserted into the database per second.
- **query** – The number of query operations per second.
- **update** – The number of document update operations per second.
- **delete** – The number of document and index delete operations per second.
- **getmore** – The number of cursor batch operations per second. This refers to batch retrieves.
- **flushes** – The number of fsync operations (flushing dirty, in-memory pages to disk) per second.
- **faults** – The number of page faults (e.g., swapping index data from memory to disk)
- **locked %** – The percent of time spent in a global write lock. (Note: This number is dependent on the data consistency requirements for the specific application).
- **% idx miss** – The percent of index access attempts that required a page fault to load an index node. This value is sampled using a rate of 100 index accesses.
- **qr | qw** – The length of the queue of clients waiting to read | write data from the deployment. This value is also a dependent of the data consistency requirements.

While the current implementation combines B-Tree index metrics with R-Tree index metrics, future versions of the MongoDB statistics tool include a separate r_idx column for monitoring performance specific to the R-Tree indexer. In the above screenshot the
A Scalable Indexing Technique and Implementation for Big Spatial Data

numbers that are followed by an asterisk refer to replicated datum. It is important to note that the statistical values depend on the data consistency requirements of the requesting client application. For example, if 100% of the clients request data with high consistency requirements (e.g., reads/writes may only be satisfied by the primary node) the locked % and qr | qw values will remain high which could impact the performance of the entire system. In reality, the implementation of a distributed deployment to service client requests with tight consistency requirements, performs significantly better than a highly optimized RDBMS deployment (and with drastically lower maintenance requirements).

6.1 Test Data Characteristics

To assess the performance benefits between the B-Tree geohash and R*-Tree multi-dimension indexer requires a significant amount of varying test data. Ideally these data sets are samples of real data the system expects, or accurate representations of expected data. The following table provides technical details for simulated spatial data sets, adapted from Beckmann and Kriegel, to assess the initial performance of the R*-Tree spatial index. Each object is within the [-180:180], [-180:180], [0:inf) bounding cube represented on a spherical earth.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number Polygons</th>
<th>Average Area</th>
<th>Variance Area</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform</td>
<td>100,000,000</td>
<td>0.036</td>
<td>9505</td>
<td>Polygonal centers follow a 3-dimensional independent uniform distributions</td>
</tr>
<tr>
<td>Cluster</td>
<td>102,400,00</td>
<td>0.0072</td>
<td>1538</td>
<td>Polygonal centers follow a distribution with 640 clusters containing ~16000 objects</td>
</tr>
<tr>
<td>Parcel</td>
<td>100,000,000</td>
<td>0.009014</td>
<td>303458</td>
<td>Decompose the unit cube into 100,000 disjoint MBRs and expand the area by a factor of 2.5</td>
</tr>
<tr>
<td>“Real”</td>
<td>120,576,000</td>
<td>0.0333</td>
<td>1504</td>
<td>MBR of elevation lines from cartography data (gathered from Kriegel 1999)</td>
</tr>
<tr>
<td>Gaussian</td>
<td>100,000,000</td>
<td>0.0288</td>
<td>89875</td>
<td>Polygonal centers follow a independent Gaussian distribution</td>
</tr>
<tr>
<td>Mixed Uniform</td>
<td>100,000,000</td>
<td>0.0072</td>
<td>6778</td>
<td>Polygonal centers follow independent uniform distribution</td>
</tr>
</tbody>
</table>

Table 1: Simulated test data sets for quantifying R*-Tree performance

To put the performance benefits to a more realistic operational test requires a wide range of spatial data. The following is a list of example data sets desired to stress operational performance:
A Scalable Indexing Technique and Implementation for Big Spatial Data

1. Large Volume LIDAR Data – ~10km² coverage, >13.5 million returns, ~0.26 m point spacing, 1.35 points/m² point density
2. Wide Area Airborne Surveillance – ~14K x 14K coverage, ½ second sampling rate
3. Multi-spectral Image Sets – 5 - 10 bands, 1-10km² coverage
4. Location-based service (LBS) positioning – locations for mobile devices (varying time samples)
5. Unstructured reports – Geocoded addresses or zip/area code resolution

6.2 Physical Infrastructure Characteristics

Figure 17 provides a high-level general illustration of a logical infrastructure layout designed to support a distributed scalable spatial indexing and analytic support system. To benchmark performance scaled across geographic locations the physical infrastructure requires a minimum of two data centers. To test operational performance and multi-level access the data centers may reside between several different physical networks and employ various security constraints (e.g., secured authentication). The physical implementation may employ devoted hardware systems or leverage scalable virtualization software configurations (e.g., Ubuntu Eucalyptus, VMWare).

Data indexing, replication, and distribution techniques are benchmarked at the task tracker and replica node software layer to assess performance at the data level within
A Scalable Indexing Technique and Implementation for Big Spatial Data

each data center. This technique eliminates the possibility of negative performance impacts due to network bottlenecks and brings the processing and performance evaluation to the physical data. Node balancing and distributed indexing is benchmarked at the shard manager and job tracker software within the analytic service layer located at each data center. This assesses data distribution performance within and between both data centers to quantify possible weaknesses within the sharding logic. Specific application implementations and distributed service requests are benchmarked and monitored within the application layer located both within and outside (e.g., mobile devices) the data centers. The performance assessment quantifies system response time from the perspective of the client application and end user.

6.3 Access Restrictions and Security Implications

Distributed index and storage systems designed to support the access, storage, and retrieval of data containing various levels of sensitivity and classification restrictions require flexible encryption, authentication, and access control implementations. A lack of access control and security capabilities in most large scale database systems often leads to a security design implementation far removed from the data source. Most often this security implementation is located at the communication points of the system using third party security software (e.g., LDAP, OAuth, firewalls, proxies) or hardware (e.g., TPM modules, disk encryption) implementations. While none of these solutions are considered incorrect, nor should they be ignored, security design implications considered at the source of the storage and indexing capabilities are critical in order to maintain optimal read/write performance of large-scale distributed data with diverse security requirements. The security controls available in the current MongoDB deployment provide database level authentication and access. While this is sufficient for controlling user and application access at the database system level finer grained security controls are often needed for system compliance with the Federal Information Processing Standards (FIPS). The Committee on National Security Systems Instruction (CNSSI) number 1253 provides significant detail for collecting, generating, processing, storing, displaying, transmitting, and receiving National Security information. Compliance with these standards, processes, and doctrines are critical for providing a system that can scale across multiple security enclaves. Implementing their well-defined security controls while maintaining the system performance necessary for large scale distributed indexing with multiple security requirements is the objective of the end system.

Cell-based access is often required at the data level for providing security access controls on a per document record basis. In most storage and indexing systems this security level is often applied well outside the implementation of the indexer. The
A Scalable Indexing Technique and Implementation for Big Spatial Data

Ramification to this approach is manifested as a significant read/write performance decrease of the indexer. The in-work approach provided by the R-Tree spatial indexer enhancement adds cell-based access to the designated KeyNode for the document. This is achieved by applying the login credentials (e.g., hashed username/password two-level authentication) as an extension to the document key.

Figure 18: Example document with 'user' tag applied to provide cell-based data access

Figure 18 illustrates a human-readable spatially indexed document (on “geo.loc” metatag) with the addition of a user tag that is applied as a secondary index to the MBR. This provides finer grained role and user-based access to specific documents stored within the distributed file system. This extension is provided in addition to existing system and network based access controls implemented at the communication layer between the mongod, mongos, and client process(es).

7 Conclusion and Recommendations

7.1 MongoDB High Scale Prototype – No Cost Contract Recommendation

The NGA lab facilities are unique, including key equipment and network connectivity with the access to critical data that can facilitate a MongoDB High Scale Prototype experiment. TST has the capability and Internal Research and Development (IR&D) resources to facilitate a MongoDB High Scale Prototype experiment at no cost to NGA. By working together on this experiments NGA and TST can Leverage commercial technology to develop advanced solutions for Big Spatial Data challenges at no cost to the government.
7.2 **MongoDB High Scale Prototype – CRADA Recommendation**

In order to demonstrate this technology TST and NGA can establish a strategic Cooperative Research and Development Agreement (CRADA). The purpose of the CRADA is that it allows NGA and TST to demonstrate a distributed spatial indexing and analytic high-scale prototype system based on TST’s “big spatial data” enhancements to MongoDB. This CRADA benefits NGA by providing hands-on access to and evaluating leading edge commercial off-the-shelf (COTS) tools and technologies. The MongoDB High Scale Prototype will optimize resources, protect the private company involved, allow both parties to keep research results, file patents, and retain patent rights on inventions developed by the CRADA.

Project goals:

- Use MongoDB with spatial enhancements, iSpatial and Ubiquity as an interface to the data to show various queries and benchmark the metrics.
- Use MongoDB with spatial enhancements, iSpatial and Ubiquity to prototype the video display over Google Earth.

7.3 **Conclusion**

TST will continue to conduct advanced research and development around the key topics discussed in this paper. It is our goal to develop a relationship with NGA where we achieve mutual benefit together. We are prepared to follow up on any topics discussed in this paper in more detail at the convenience of NGA.