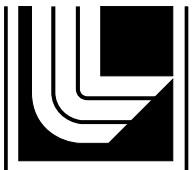

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-225827

Evaluation of Potential Large Synoptic Survey Telescope Spatial Indexing Strategies

Sergei Nikolaev, Ghaleb Abdulla, Robb Matzke

November 3, 2006

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Evaluation of Potential LSST Spatial Indexing Strategies

Sergei Nikolaev, Ghaleb Abdulla, Robb Matzke (LLNL)

Abstract

The LSST requirement for producing alerts in near real-time, and the fact that generating an alert depends on knowing the history of light variations for a given sky position, both imply that the clustering information for all detections is available at any time during the survey. Therefore, any data structure describing clustering of detections in LSST needs to be continuously updated, even as new detections are arriving from the pipeline. We call this use case "incremental clustering", to reflect this continuous updating of clustering information. This document describes the evaluation results for several potential LSST incremental clustering strategies, using: 1) Neighbors table and zone optimization to store spatial clusters (a.k.a. Jim Grey's, or SDSS algorithm); 2) MySQL built-in R-tree implementation; 3) an external spatial index library which supports a query interface.

1. Some Relevant LSST Numbers

- Number density of galaxies: 30/sq. arcmin (this number refers to the density of objects in individual exposures; object density in stacked images is 50/sq. arcmin)
- Sky area covered by 1 exposure: 10 sq. degrees
- Number of detections measured per exposure (static): ~1.5 million
- Number of detections measured per exposure (DIA): ~100,000
- Number of unique fields in the sky: ~2000
- Number of times each field is observed in 1 year: ~100
- Number of exposures per minute: ~4
- Length of the survey: 10 years

2. Evaluation Hardware

The tests described here were run on a dedicated server (hydra.ucllnl.org) with the following characteristics:

- CPU: 4x dual-core Opteron 870 (2GHz)
- RAM: 4Gb (4x1) PC3200 DDR400
- Disk space: 8x400GB Hitachi SATA HDD on 3WARE 9500-S 8-port RAID controller. Disks are in JBOD configuration (no RAID), mounted on /d1 - /d7, with one "system" disk (mounted on /).
- Disk space (additional): 2x LaCie Bigger Disk, 1.6TB each connected through Firewire 800 interface
- OS: Linux RHEL 4 AS Edition 64-bit (2.6.14.3 kernel)

The server was used exclusively for testing, in single-user environment and no other user-run processes.

3. Evaluation Software

- Database engine: MySQL v.5.0.21, x86-64 version. RDBMS is configured as 7 separate servers (mysql-max) running concurrently, each with its own data directory (e.g. /d1/mysql/data) and a socket file (e.g. /d1/mysql/hydra.sock). For our testing, we used a single DB server.
- Spatial index library (SaIL; ver. 1.0). Original version of the SaIL and the supporting *tools* library are due to Marios Hadjieleftheriou. The software was modified by Robb Matzke to support MPI-based parallel operations, and implement various algorithm enhancements.
- MPICH2 (ver. 1.0.4)

4. Evaluation Results

4.1 Spatial Index Library (SaIL)

The test suite for the library is located in the subdirectory *regressiontest/rtree*. It includes executables both for loading (creating) the R-tree index, and for querying it. The library expects Cartesian input data in [0, 1] × [0, 1]. Because of this, we could not run tests on precursor SuperMACHO dataset, but rather used a synthetic dataset created by the *Generator* utility included in the distribution. The 'default' input data file contained 1.5 million rows, to approximate the number of objects coming out the LSST (static) pipeline every 10-15 sec. The file size was varied, for testing how index creation scales with the data size. The data were formatted as points, i.e. (*x, y*) pairs, rather than Minimum Bounding Rectangles (MBR), which required a minor modification of the original code.

Below is a typical result of creating an R-tree by RTreeLoad:

- Input rows: 1,500,000
- Time elapsed: 5 min 27 sec (1 CPU)
- Tree height: 4
- Leaf capacity (fan-out factor): 100
- Number of nodes: 22,518
- Pages/level: 1 – 5 – 327 – 22185
- Node utilization: 67%

The CPU utilization for loading is close to 100%, and the memory impact is very small (<1%). For each input file, the library creates 2 files, index and data file, both in binary formats. For a 41Mb ASCII input file, the sizes of the newly created files are 103Mb (data) and 803Kb (index). In its original form, SaIL is not capable of updating an

existing index (specifying an existing tree file overwrites the file, instead of appending to it).

The R-tree building time as a function of the input file size is shown in Table 1. The index creation time is proportional to the size of the input file (should we see NlogN?). The sizes of the created data and index file also scale linearly with the size of the input file.

Number of rows, input file	Index file size, KB	Data file size, MB	R-tree build time, sec
1.0e4	4.5	0.57	2
5.0e4	23.5	3.0	10
1.0e5	52.2	6.7	19
7.5e5	398.2	50.9	152
1.5e6	803.3	102.9	304
3.0e6	1604.7	206.0	610
6.0e6	3215.4	411.6	1207

Table 1. Ingest results for spatial index library (SaIL), as a function of the input file size.

Scaling the size of the index file to LSST data volume, we expect combined index size 1MB/image * 2000 images/night * 300 night/year * 10 years = 6 TB, i.e. it should be possible to store this index in RAM by 2013. If we consider only DIA detections (the ones which will generate real-time alerts in LSST), the combined index size drops to 330GB, which will fit into RAM of a medium cluster (~20 nodes) even today.

The fan-out factor (leaf capacity) is a user-defined parameter for the library which has an impact on performance. The load time as a function of leaf capacity is shown in Table 2. Results indicate that smaller fan-out factors produce larger data and index files. The time to create the index, on the other hand, is a non-linear function of the leaf capacity. These results suggest that with constraints on load times and available RAM an optimal fan-out factor exists for a given dataset.

Fan-out factor (leaf capacity)	Index file size, KB	Data file size, MB	R-tree build time, sec
10	8643.0	1106.3	324
30	2715.0	347.5	185
100	803.3	102.9	304
1000	160.3	51.4	2183

Table 2. Ingest results as a function of the tree fan-out factor (leaf capacity).

The test suite for SaIL also includes the parallel version of the tree loading algorithm, RTreeLoadParallel. The load times for our 1.5 million row LSST file as a function of number of processes N are summarized in Table 3. The results show that for small number of processes (N<5), the parallel version scales linearly with the number of CPUs.

After that, the improvement slows down. After a physical CPU capacity of the hardware is reached (N>8), we see no improvement, as expected. The reason could be a particular dataset we are dealing with, or competing disk I/O requests from processes.

Note that one CPU in RTreeLoadParallel is reserved for a "master" process which does not contribute to creating the R-tree index, so the number of processes actually working on creating R-tree is always N-1 (this is verified by running "top" command: only N-1 copies of RTreeLoadParallel are visible). Each process creates its own pair of data and index files.

Number of processes, N	R-tree build time, sec
1	–
2	294
3	140
4	116
5	80
6	71
7	70
8	53
9	50
10	53

Table 3. Parallel R-tree index create time as a function of the number of CPUs.

Querying R-tree is implemented as single-threaded (RTreeQuery) and parallel (RTreeQueryParallel) versions. The single-threaded version only works with tree files produced by single-threaded data loading utility, RTreeLoad, while the parallel version only works with trees created by RTreeLoadParallel. Moreover, the number of processes to use for RTreeQueryParallel must be the same as the number of processes (N) used for RTreeLoadParallel (actually, using N-1 processes also works for RTreeQueryParallel).

For our test query, we searched for all rows which intersected a moderately large rectangle, [0.4, 0.6] × [0.4, 0.6]. For a single-threaded version, the query returned the expected set (39648 rows) in 1 sec. Parallel version returned the same set of rows, and also took about 1 sec (no gain for parallel querying?)

To summarize, our tests for the spatial index library are quite promising: the ingest scales linearly with input number of rows and the parallel version of the R-tree index scales almost linearly with the number of CPUs doing the job. The expected combined index file for LSST detections is small enough to fit in RAM by 2013, which will speed up clustering for new detections. For DIA files (100,000 detections per image), the R-tree index files will fit into memory of a medium-sized cluster even today. The index build time for one DIA file, ~20 sec, is comparable to the ~15 sec interval between image arrivals. This suggests that with a minimum parallelization, this task is doable even today. Our tests showed that for a given dataset, there is an optimal choice of the fan-out factor for the tree. Finally, querying the tree is very fast, even though there is no advantage for using parallel queries. The query time depends only on the depth of the

tree, and should be approximately constant for all queries covering equal areas (true or false? Would be nice to time the query on a real astronomical object).

4.2 SDSS Algorithm

SDSS algorithm tackles clustering problem by building a "Neighbors" table, which lists all detections within a certain distance from a given detection. In its simplest form, each row of the Neighbors table consists of ID1 for primary star, ID2 for neighbor, and R for distance between the primary and the neighbor, so the smallest row size is 20 bytes (2 bigint, 1 float). In SDSS the Neighbors table holds all pairs or detections within ½ arcmin from each other. To create it, one uses "Zone" table (cf. Section 4.4 of Gray et al. 2004a):

```
INSERT Neighbors
SELECT o1.objID AS ID1, o2.objID AS ID2, …
FROM zone o1 JOIN zone o2 ON o1.zoneID-@deltaZone = o2.zoneID
     -- @deltaZone is in {-1, 0, 1}
     AND o2.ra BETWEEN o1.ra-@r/(cos(radians(o1.dec))+@eps)
                   AND o1.ra+@r/(cos(radians(o1.dec))+@eps)
     -- cos() accounts for spherical geometry
     -- @eps to prevent division by zero at poles
WHERE ((o1.ra >= 0 AND o2.ra < 360) OR (o2.ra >= 0 AND o2.ra < 360))
     -- make sure not both objects are in marginal zone near zeroth
     -- meridian
      AND o1.objID < o2.objID
     -- selects only 1/2 the objects; SDSS later adds a mirror
     -- image of each pair
      AND o2.dec BETWEEN o1.dec-@r AND o1.dec+@r
     -- declination condition
      AND 4*power(sin(radians(@r)),2) >
         power(o1.x-o2.x,2) + power(o1.y-o2.y,2) + power(o1.z-o2.z,2);
     -- careful distance filter
```

In the query above, @r is the proximity radius (1/2 arcmin for SDSS). Zone table is essentially the table of detections, which has one extra attribute, zoneID. zoneID is basically the declination of the detection divided by some @zoneWidth, expressed as an integer:

$$\text{zoneID} = \text{floor}((\text{dec} + 90.0)/@\text{zoneWidth}).$$

For example, if @zoneWidth = 1/3600 degrees = 1'', zoneID will give the number of whole arcseconds to the object's declination, counted from the South Celestial Pole.

The SDSS algorithm proceeds by populating the Neighbors table, selecting a subset of it using proximity radius 1'' (called Match table) and then running an iterative procedure on the Match table to ensure that all detections in a cluster form a complete graph (i.e. all members are connected to all others). After the Match table has fully connected clusters,

the final table, MatchHead names the clusters by the lowest objID in each cluster, and also computes some average properties of the cluster, e.g. position, variance, etc. For a full description, see Gray et al. 2004b.

The procedure is straightforward to implement in SQL, using e.g. stored procedures. Unfortunately, it probably will not work for *incremental* clustering. Let's consider the sequence of steps that must be completed in order to find all neighbors for a new detection:

1. Query MatchHead table using coordinates (RA, Dec) of the new detection, and retrieve objID for the cluster closest to the position;
2. Query Match table for the retrieved cluster objID, and extract all objIDs for detections in the fully connected cluster. (Note: objID in 1 and 2 denote different things; the former is a cluster ID, the latter is a detection ID.);
3. Query Detections table given objIDs of cluster members to extract their measurements (steps 1-2-3 can be written as a single join query) ;
4. Update Neighbors table and its index to add pairs linked to the new detection;
5. Update Match table and its index;
6. Update MatchHead table and its index.

This sequence must be repeated for each new detection coming in, and everything must be completed in ~15 seconds, before next image comes in. Steps 1, 2, and 3 are relatively easy, and can be completed in a short time, given appropriate indexes on MatchHead, Match, and Detections tables. The problem lies in steps 4, 5, and 6, which requires inserting new data to three large tables, and updating indexes on each of these tables. Let's consider some numbers (for DIA mode only, because real-time requirement for clustering is not critical for object-based photometry):

1. At 100,000 DIA detections per 10 square degrees, an average separation between the DIA detections is $n^{-1/2} = 0.6$ arcmin. Adopting cluster radius of ½ arcmin in Neighbors table, and ignoring moving objects (this is the best possible case), each new image adds a single detection to each cluster in the field every 15 sec.

2. Assuming all DIA objects are stationary, we have 100,000 clusters in an LSST field. Then, at any given instant, the number or rows in the Neighbors table for this field is $N_{rows} = 100{,}000*n*(n-1)$, where n is the number of epochs (or times) this field has been observed. Here we assume that Neighbors table holds all pairs, including mirror ones, otherwise N_{rows} should be divided by 2. In one year, LSST will have accumulated n ~ 100 epochs, and the number of rows in the Neighbors table (for one LSST field only!) is already nearly 1 billion. The total disk space required to accommodate Neighbors tables for all LSST fields, for 10 years is 2000 LSST fields * 10^{11} rows/field * 20 bytes/row = 3.6 Petabytes.

3. The storage size for an index is evaluated as $N_{rows}*3/2*$(idx_len+data_ptr_len), see MySQL Reference Manual, Ch. 7.2.2. At the end of LSST survey, we saw that $N_{rows} \sim 10^{11}$. Assuming Neighbors table has an index on (ID1, ID2), index

length (idx_len) is 16 bytes (2 bigint), and data_ptr_len is 8 bytes, so this index requires storage of ~ 3Tb. For all 2000 LSST fields, the index size would be ~6Pb. This index needs 20000 times more RAM, compared to the combined index size for SaIL (330GB)! With this much RAM required it's probably impossible caching this index even in 2023, which means extra disk I/O to read/write the index during the survey.

4. Everything said so far also extends to Match table, as well. Despite the fact that Match table is built using smaller cluster size (1'' instead of ½ arcmin), the clusters in Neighbors table are really sub-arcsecond size (expected astrometric accuracy of LSST). This means both the disk space, and the RAM estimates have to be doubled.

5. MatchHead table is a much smaller one, so it does not contribute significantly to the total required disk space and RAM. However, the cluster averages it holds must be updated with every new image. This means, 100,000 rows must be updated every 15 sec. The index must also be updated in the same time.

6. The estimates above are lower bounds on the actual requirements, since many LSST detections will be due to moving objects (as much as 50%, depending on the field). Movers will "link" nearby clusters together, effectively doubling the cluster size, leading to even more Neighbors and Matches.

To summarize, RAM required to store the indexes will likely not be available in 2013 (or even 2023). More importantly, maintaining up-to-date Neighbors, Match, and MatchHead tables and their indexes require a large number of disk I/O ops, making it impractical as an incremental approach.

4.3 Native MySQL R-Tree Implementation

MySQL has a built-in implementation of R-tree index, to support spatial queries. We examined the performance of MySQL R-tree and B-tree indexes by implementing a test database and populating it with real SM data:

```
CREATE TABLE TestBtree ( -- To test B-tree index performance
    id bigint,
    mjd double,
    ra double,
    decl double,
    x double,
    y double,
    z double,
    zone bigint
);

CREATE TABLE TestRtree ( -- To test R-tree index performance
    id bigint,
    mjd double,
```

```
    p POINT NOT NULL,
    ra double,
    decl double,
    x double,
    y double,
    z double
);
```

For faster ingest, we use bulk load from file (which requires a special trick for spatial data type):

```
LOAD DATA INFILE '…' INTO TABLE TestBtree
FIELDS TERMINATED BY ' '
(id,mjd,ra,decl)
SET x = cos(radians(decl))*cos(radians(ra)),
        y = cos(radians(decl))*sin(radians(ra)),
        z = sin(radians(decl)),
        zone = floor((decl+90)*3600);

LOAD DATA INFILE '…' INTO TABLE TestRtree
FIELDS TERMINATED BY ' '
(id, mjd, ra, decl)
SET p = PointFromText(CONCAT('POINT(',ra, ' ', decl, ')')),
        x = cos(radians(decl))*cos(radians(ra)),
        y = cos(radians(decl))*sin(radians(ra)),
        z = sin(radians(decl));
```

The first query ingests 34,590,801 rows in 2 min 50 sec; the second version ingests the same number of rows in 6 min 53 sec. After the data are ingested, we create corresponding indexes on each table. For TestBtree, we use a composite index on (zone, ra), and for TestRtree we use a spatial index on its Point attribute:

```
CREATE INDEX idx_zone_ra ON TestBtree (zone, ra);

CREATE SPATIAL INDEX idx_p ON TestRtree (p);
```

We will refer to these as "main" indexes hereafter. It takes 7 min 9 sec to create the composite B-tree index (367Mb index size), and much longer (6 hours 36 min 53 sec) to create the spatial R-tree index (2.1Gb index size).

To compare the performance of the two indexes, we search for neighbors using the following two procedures:

```
drop procedure if exists fetchNeighborsBtree;
delimiter //
create procedure fetchNeighborsBtree (IN objid bigint)
begin
  declare rad double;
  declare x0 double;
  declare y0 double;
  declare z0 double;
  declare r0 double;
```

```
  declare d0 double;
  declare zone0 bigint;
  select ra,decl,x,y,z,zone into r0,d0,x0,y0,z0,zone0
  from TestBtree where id=objid;
  set rad = 1/3600;
  select * from TestBtree
  where zone between zone0-1 and zone0+1
    and ra between r0-rad/cos(radians(d0)) and r0+rad/cos(radians(d0))
    and degrees(acos(x*x0+y*y0+z*z0))<rad;
end;
//
delimiter ;

drop procedure if exists fetchNeighborsRtree;
delimiter //
create procedure fetchNeighborsRtree (IN objid bigint)
begin
  declare rad double;
  declare x0 double;
  declare y0 double;
  declare z0 double;
  declare r0 double;
  declare d0 double;
  declare r1 double;
  declare r2 double;
  declare d1 double;
  declare d2 double;
  declare box geometry;
  select ra,decl,x,y,z into r0,d0,x0,y0,z0
  from TestRtree where id=objid;
  set rad = 1/3600;
  set d1 = d0 - rad;
  set d2 = d0 + rad;
  set r1 = r0 - rad/cos(radians(d0));
  set r2 = r0 + rad/cos(radians(d0));
  set box = GeomFromText(concat('Polygon((',
    r1,' ',d1,',',
    r2,' ',d1,',',
    r2,' ',d2,',',
    r1,' ',d2,',',
    r1,' ',d1,'))'));
  select * from TestRtree
  where MBRContains(box, p) and degrees(acos(x*x0+y*y0+z*z0))<rad;
end;
//
delimiter ;
```

Both procedures take a single input parameter, a detection ID, and return all rows within 1 arcsecond from that detection. For fetchNeighborsBtree, we narrow down the search area by using conditions on zone and ra, while for fetchNeighborsRtree we use Geometry function MBRContains(). The final condition in the WHERE clause is the exact test for angular distance.

To speed up both procedures, we need an additional index (B-tree) on id attribute, to extract query parameters quickly. This index was added after the main indexes were

already built. Despite the identical number of rows in both tables, the time required for creating this index is significantly different: for TestBtree index creation takes about 10 minutes, while for TestRtree it takes more than 16 hours! Monitoring system resources during the index creation on TestRtree showed that disk I/O operations were maxed out, probably because the DBMS needs to put each id into a proper place within the existing R-tree index structure, which requires a lot of disk seeks. Since we did not see this behavior for B-tree indexes, it leads us to believe that indexes of different types, R-tree and B-tree, on a single table do not "mix" well. Despite the markedly different performance, adding an extra index had a similar effect on the existing index sizes: the B-tree index size went from 367Mb to 874Mb, while R-tree index size increased from 2.1Gb to 2.6Gb.

Both procedures were called for a number of identical object IDs ("clusters"), and the results were timed. Performance is summarized in Table 4.

Object ID	Rows returned	Time in sec, fetchNeighborsBtree()	Rows returned	Time in sec, fetchNeighborsRtree()
1	98	21.69	98	1.08
10	84	0.69	84	0.09
1000	1	5.79	1	0.05
10000	6	32.97	6	0.26
100000	76	11.56	76	0.92
1000000	3	24.24	3	0.08
10000000	34	29.34	34	0.00

Table 4. Performance of MySQL B-tree and R-tree indexes, for fetchNeighbor() queries.

The results indicate that both procedures are consistent with each other, i.e. return the same neighbors for each "seed" detection id. Also, the R-tree procedure is typically 1-2 orders of magnitude faster than the B-tree procedure. Note that this performance gain is not due to result caching (fetchNeighborsRtree was run after fetchNeighborsBtree), as it is observed even for random (non-overlapping) ids.

From our testing native MySQL implementation of B-tree and R-tree indexes we conclude that about every aspect of working with R-trees takes longer than the corresponding operation with B-trees, with the lone exception being the actual querying. In particular,

1. Ingest into R-tree table takes ~**2.5 times longer** for the same number of rows than the ingest in B-tree table, probably due to a need to convert text to geometry;
2. Building an R-tree index on Points takes ~**50 times longer** than a composite B-tree index on (zone, ra);
3. The size of the R-tree index (2.1Gb) is ~**6 times larger** than the size of the B-tree index (367Mb);
4. Adding a new B-tree index to a table takes ~**100 times longer** for table with existing R-tree index than with existing B-tree index (this may scale better for smaller tables?)

5. Query on a table with R-tree index is **10-100 times faster** than the same query using composite (zone, ra) B-tree index.

Combined, these results indicate that the choice of B-tree vs R-tree should be dictated by the frequency of expected index and data updates, as well as the expected query load. For real-time clustering problem relevant to LSST, updates are critical (we need to have an up-to-date information in Detection table), which probably means B-trees are preferable. Compared to an external spatial index library (see above), neither of native MySQL indexes is impressive.

P.S. May need to do additional tests on updating an indexed table (B-tree index vs. R-tree index). So far, we bulk loaded into an empty table and then build an index. Also, how does performance scale with the table size?

4.4 Alternative Clustering Strategies

- HTM (external library) – tests by SDSS DB group indicate that an external HTM library performs poorly as compared to the internal MSSQL implementation of HTM. In turn, internal HTM algorithm in MSSQL is worse than the "zoning" approach the group used.

- HTM on-the-fly (Brunner). Unlikely a contender, because implemented in Java, and is essentially the same HTM algorithm that SDSS claims is slow outside a database.

- OPTICS algorithm allows a representation of spatial data in such a way that clusters are easy to extract.

5. Conclusions

Of the three potential clustering technologies, at this point most promising is an external spatial indexing library (SaIL). With parallelization and some modifications (e.g. extending to spherical coordinates; allowing tree file updating, etc), this technology could support real-time detection clustering for LSST.

This work was performed under the auspices of the U.S. Department of Energy, National Nuclear Security Administration by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

1. Gray, J. et al. “There Goes the Neighborhood: Relational Algebra for Spatial Data Search”, 2004a, Tech.Rep. MSR-TR-2004-32

2. Gray, J. et al. “Computing the Match Table”, 2004b

3. MySQL Reference Manual, *http://dev.mysql.com/doc/refman/5.0/en/index.html*