Time Series

Scale

The LSST survey will visit any patch of sky visible from Cerro Pachon every 3 days on average for 10 years, detecting some 40 billion astronomical Objects, each with approx 1000 data points, as shown in the below table.

catalog	first data release	last data release
Source	~2.5E+11	~5.7E+11
FourceSource	~4.9E+12	~3.2E+13

Query complexity

The main types of time-series based data analysis are:

- Object type classification
- Similarity search
- Outliers detection

Astronomical object classification

This task aims to extract the nature of an observed object from its time series data. It is something that should be performed upfront the database through the main pipeline, but, of course, time series can be reanalyzed with different algorithms. Some recent papers present a new approach on this side leveraging machine learning techniques:

- ?QUASI-STELLAR OBJECT SELECTION ALGORITHM USING TIME VARIABILITY AND MACHINE LEARNING: SELECTION OF 1620 QUASI-STELLAR OBJECT CANDIDATES FROM MACHO LARGE MAGELLANIC CLOUD DATABASE (Dae-Won Kim, Pavlos Protopapas, Yong-Ik Byun, Charles Alcock, Roni Khardon and Markos Trichas)
- ?OPTIMAL TIME-SERIES SELECTION OF QUASARS (Nathaniel R. Butler and Joshua S. Bloom)

The target of time series analysis is the classification of objects into a limited number of bins:

- QSO
- Microlensing
- Eclipsing binaries
- Supernovae
- Variable stars (RR Lyraes, Cepheids)
- Long period variables
- Be stars
- ...

Objects that do not fit this classification are good candidates for outliers investigation and similarity search analysis.

Similarity search

Similarity search in the LSST catalog means, given a query object, being able to answer the following three queries:

- Find all the similar objects within a range (**range search**)
- Find the k most similar (or closest in the parameter space) objects (**kNN search**)
- Return the objects in order of similarity, possibly within a range (**rank search**)

Outliers detection

This part is under development by the ?Kirk Borne's group at George Mason University. They have developed a data structure that implements the search hierarchy and the related algorithms know ad kNN-DD.

Index dimensionality and other requirements

Expected complex queries often include comparing time series. It is worth noting that different groups/astronomers will often have their own way of defining what "similar" means. Different classes of queries will often rely on different subset of dimensions. We expect we will need to maintain a highly multi-dimensional index (perhaps with as many as 10, 20 or 30 dimensions).

Dimensions that will likely be useful when dealing with time series include:

- number of data points
- period
- short flux variations (e.g., daily)
- long flux variations (e.g., yearly)
- mean flux variations

The above will likely need to be captured for all-colors, as well as per-color. The level of uncertainly further complicates the problem.

Based on the LSST requirements, the indexing structure can be static (built once, read only).

Many systems supporting time series analysis deliver approximate results, and they allow user to do a trade-off between accuracy and latency. To meet the LSST requirements, we are not planning to rely on approximate results.

Research

There are two main branches of research about similarity search in high-dimensional spaces:

- Embedding methods
- Distance-based indexing methods

The distance function

Very often in high dimensional spaces the only reasonable piece of information that is available is the inter-object distance. Similarity is commonly expressed by means of a distance metric that could be

computationally intensive, especially with growing number of dimensions.

Embedding methods

This techniques are based on dimensionality reduction, the high-dimensional space is reduced (embedded) into a considerably less-dimensional space deriving some *features* based on the inter-object distance from the actual dimensions. The search is performed in the less-dimensional space using a different distance function (that is generally contractive) that results less computationally intensive and then refined in the original space. Often the two phases can be parallelized. A good survey of the embedding methods is: ?Searching in High-dimensional Spaces -Index Structures for Improving the Performance of Multimedia Databases (Christian Böhm, Stefan Berchtold, Daniel A. Keim)

Distance-based indexing methods

This technique assumes that the only available information is the distance function and uses it to index the data objects using some of them as distance references. The advantage of these methods is the reduction of computations needed to perform the search once the indexing structure has been constructed. Distance-based indexing methods can be confined to two large macro-families:

- Ball partitioning
- Generalized hyperplane partitioning

The former recursively partitions the space using hyperspheres around a *pivot* point until reaching a *ball* size that is small enough to be fully searched. The latter recursively splits the space using two *pivot* points and assigning the objects to distinct subspaces based on the relative distances to the pivots, essentially defining a splitting hyperplane. A good survey of the distance-based indexing methods is ?Index-Driven Similarity Search in Metric Spaces (Gisli R. Hjaltason, Hanan Samet) This branch of similarity search seem to be the most promising for the application to the LSST catalog and most of the research has focused on it. In particular the ball partitioning category has been investigated.

The search hierarchy

The distance-based methods inducts a *search hierarchy* on which common search algorithms can be used. The hierarchy is generally backed up by an index structure that is built differently by the individual methods.

Static and dynamic structures

Distance-based indexed methods were born as purely static, it means that once the index has been built it is immutable and any change to the multidimensional space implies the rebuild of the whole structure. This is generally a very time-consuming operation. If multiple distances are used to segregate the search for each category of objects, multiple search indexes must be constructed. The reference static indexing structure is the ?VPtree (Peter N. Yianilos). Although some variants of the static methods have been developed to add mutability support to the index structure, some other structures have been designed to operate over changing metric spaces. One of this is the ?Mtree (P. Ciaccia, M. Patella, P. Zezula) that has been designed for large data structures that can't be entirely cached in memory and have a large disk footprint.

Approximate search methods

Full exact similarity search on large data sets can be quite slow even using a well designed indexing structure, for this reason approximate search research have ramped up during the last few years, given the growth rate of

the data sets. Approximate search may, depending on the strategy, prevent *false dismissals* as well as restrict the error margin to a certain value. Queries may return an order of magnitude (or more) faster than exact search. Considering the uncertainty of the data points in the metric space and the additional investigation needed on the query results, this approach is worth to be considered. A good survey of approximate search indexing structures and algorithms is ?Approximate similarity search: A multi-faceted problem (M. Patella, P. Ciaccia)

KD-trees vs R-trees

- Relevant page: ?http://stackoverflow.com/questions/4326332/could-anyone-tell-me-whats-the-difference-between-kd-tree-and
- in summary, r trees seems better, because they are disk-oriented. In our case, difficulty with updating kdtrees does not matter, we always bulk-load data

Thoughts on Time Series directly in MySQL

MySQL has recently reworked their GIS support. GIS is based on spatial indexes, which under the hood uses R-trees with quadratic splitting (more at ?optimizing spatial analysis).

The spatial support in mysql is fully supported in MyISAM. InnoDB engine supports spatial data types, however it does not support spatial indexes (as of mysql 5.6, May 2013).

One option would be to rely on the built-in R-tree indexes. This would require extending data types, as the existing spatial data types are unlikely to be sufficient. Also, the quadratic splitting implementation is likely insufficient to handle multi-dimensional space we need to cover.

It is worth noting that PostgreSQL database has fully integrated similarity search into the SQL engine using dedicated indexing structures (?Postgresql 9.2 SP-GiST)

Discussion about architecture

The baseline architecture for the LSST database relies on qserv. It is natural for the time series system to rely on qserv as well. Since time series of individual objects are not correlated, the qserv architecture can be very easily used without any special modifications, or even dealing with any overlaps.

The most challenging aspect is selecting the *best* indexing structure for the search hierarchy, and the selection of an efficient method to store it.

The two indexing structures we considered:

- architecture based on specialized library, with custom format
- fully custom index, in standard RDBMS

Specialized-library-based architecture

The proposal focuses on relying on off-the-shelf library for approximate similarity search, available as open source under the BSD license, called ?FLANN. It is integrated in many free and commercial products.

The FLANN library

FLANN can be described as a system that answers the question, ?What is the fastest approximate nearest-neighbour algorithm for my data??. FLANN takes any given dataset and desired degree of precision and use these to automatically determine the best algorithm and parameter values. Under the hood the library exploits a new algorithm that applies priority search on hierarchical **k-means trees**, which has been found to provide the best known performance on many datasets. After testing a range of alternatives, the FLANN developers have found that **multiple randomized k-d trees** provide the best performance for other datasets. FLANN is able to automatically detect the ideal algorithm and the best parameters to build the index and minimize query times. The cost is computed as a combination of the search time, tree build time, and tree memory overhead. Depending on the application, each of these three factors has a different importance (controlled by two factors build time weight, wb, and a memory weight, wm).

FLANN is a C++ library (current version is ?1.8.4) with C, Matlab and Python bindings. With version 1.8.x the library is capable of:

- Generating multiple instances of the index object
- Dump and reload the index to/from a file
- Incrementally add and remove points to/from indexes
- Spawn MPI powered searches
- Using OpenMP multi-threading
- GPU kd-tree matching for 3D features on CUDA compatible hardware

Scalability tests

For these test the Python wrapper has been used, it is affected by some limitations:

- No multi-threading
- No incremental addition/removal of points
- No CUDA support

Nevertheless some of the results are quite interesting.

The tests have been performed producing a Numpy array Nx3 with N ranging from 1.000 to 1.000.000.000. The latter limit has been imposed by the amount of RAM on the server (currently 192GB). The index has been created using an hierarchical k-means decomposition. The results are summarized by the following graphs:

MySQL-based architecture

Maintain a rich set of columns that represents time series for each object in an Object-like table (that is, maintain a table with one row per each time series), with things that will be needed for time series analysis. It probably makes sense to have just one such table for both Source and ForcedSource.

Maintain a set of indexes on this table (let's call it "TimeSeriesSummary"), possibly composite indexes too, and use these indexes for very lightweight analysis that can be trivially answered through these indexes.

Do not maintain any indexes on Source/ForcedSource tables, except just one index on objectId (or equivalent) used for rapid location of individual time series.

In normal mode, do not run shared scans on Source/ForcedSource tables. Instead:

1. continuously run shared scans on the "TimeSeriesSummary" table.
2. use all the IO from disks that hold Source/ForcedSource tables to answer point queries, something like:

```
SELECT ...
FROM Source
WHERE objectId IN (
    SELECT objectId
    FROM TimeSeriesSummary
    WHERE ...     )
```

Since TimeSeriesSummary table is compact, we could have rapid turnaround (~hour or faster). We could select millions of individual time series per hour, (quick rough estimate: data for Source + ForcedSource in DR11 = 4.7 PB, that is ~500 10TB drives, assuming 100 random fetches per sec, theoretically we could fetch 20-30K time series per second, or ~100 million/hour if we used all spindles all the time)

What about queries that select lots more time series than we can handle using the approach outlined above? Periodically, (say once per month?) we disable scan on TimeSeriesSummary table & fetching individual time series, and instead we do one shared full table scan on Source/ForcedSource tables, and we add any information to the TimeSeriesSummary table that is necessary to answer new type of queries, including queries that would otherwise select too many time series.

Advantages of the above approach:

- it is much more **scalable**. Supporting 20-30, or even 50 columns in the TimeSeriesSummary table is trivial. 20-30 or 50-d spatial index is not.
- it is easily **extensible**: we can add new summaries by adding new columns to TimeSeriesSummary with no problem. Adding a new dimension to R-tree or even worse KD tree is very heavy
- it improves **performance**: it reduces turnaround time from day or week to an hour without any extra cost/resources
- it is **easy to maintain**: it fits naturally into our current qserv model. It is based on very well understood technology, every DBMS has the features we rely on
- it is **well understood**: it can be handled through SQL, there is no need to come up with new interfaces
- it is **simple**: it certainly much simpler that multi-dimensional R-trees and KD-trees etc...

Disadvantages?

- if we don't have the appropriate information in the TimeSeriesSummary table, the analysis will have to wait until we add it there.

Related reading

- ?Optimal Time-Series Selection of Quasars
- ?A Bayesian method for the analysis of deterministic and stochastic time series
- ?Improved methodology for the automated classification of periodic variable stars