Efficient 3D and 4D Geospatial Indexing in RDF Stores with a Focus on Moving Objects

Steve Haflich, Jans Aasman

Franz Inc., 2201 Broadway, Suite 715, Oakland, CA 94612
smh@franz.com, ja@franz.com

Abstract.

An RDF (Semantic Web) Database is a thin layer on top of a Directed Graph Database. Data stored in a linked graph is very different from data stored in a table, and the storage has different strengths. AllegroGraph RDFStore is a high-performance, persistent RDF graph database. In this paper, we first present AllegroGraph's existing capabilities for 2D proximity queries using geospatial indexing. This leads to an important digression – our position on geo encoding and serialization. We then describe our ongoing work extending to 3D and 4D to support moving objects (MOBs) with efficient cursors on MOB tracks. Finally, we posit five classes of MOB queries with increasing difficulty.

Introduction

Combining geospatial and temporal reasoning in a single query framework seems an important capability for the Semantic Web. Applications are obvious for the intelligence community, for logistics control, for social networking applications, and elsewhere. An application that must reason efficiently about events that happen in and paths that traverse through both time and space must have capabilities that can deal with them together.

We see an active interest in geospatial computing in the Semantic web community. A prominent example is the Linked Data initiative that has the geonames database as one of the central data sources pointed to by many other data sources. The geospatial community also shows more and more interest in RDF and semantic technologies. The National Map project is one of the examples where we see how the experts in the field try to come up with ontologies that will allow better use of the information in various data sources.

But curiously, there is hardly any interest in dealing with time in the Semantic Web community. The W3C site shows no activity and the temporal reasoning community rarely considers RDF.

It follows that there is also not much interest apparent in the combination of geo-spatial and temporal reasoning in one computational framework. Recently in January of 2009, the National Science Foundation held an expert meeting to gauge whether there is interest in the academic arena to do some more research in this area.

The AllegroGraph RDF database has supported basic temporal reasoning and 2D geospatial querying for two years, but recently we engaged projects that pushed the representation requirements in unexpected ways. One project deals with AIS vessel-tracking data and the other deals with GPS tracking data for cell phone users. The current separate facilities for 2D geospatial indexing and temporal indexing could not in combination provide sufficient efficiency to deal with moving objects (MOBs) on a large scale. Consequently, we started a research project for direct indexing of 3D and 4D data. This paper reports some of our findings and ongoing development.

A Position Statement about Geo External Syntax

We have several observations about data representation. It is useful to separate the issues of internal representation from external representation(s), i.e. serialization formats.

It is frequently the practice, probably inherited from relational databases, of storing longitude, latitude, optionally altitude and time, each in a separate triple. These correspond to the separate columns in a RDB. We show below that this is grossly inefficient for locality queries, and can result in O(n^2) or worse performance comparing the size of the database against the number of data in the result set. Our experience is that longitude/latitude and longitude/latitude/time (etc.) should be encoded and indexed internally as a single RDF quantity. (It is essentially never the case that one wants to retrieve or search any one of these data without retrieving them all together!) We have devised and will describe an indexing scheme that supports locality search with near-linear O(n) time in the number of entries in the result set.

If geo or geo/time data should be combined into a single datum internally, they should also remain a single datum when externalized, e.g. in N-triples format. RDF serialization places no requirements on triple ordering, and if a latitude-longitude pair was to become hugely separated in serialization it could be pernicious to have to recombine them during deserialization. Therefore we make the following position statement: There should be standard RDF types for externalized geo data that allows geo and geo/time entry to be represented as a single lexical string. As a specific straw proposal, externalization for geo position could be something like ISO6709, and externalization of a MOB datum should be something like the concatenation of an ISO6709 string with an ISO8601 string. (The details are important, of course, and remain to be worked out.)

How We Do 2D Geospatial in AllegroGraph

This section will explain the principles that underlie our 2D, 3D, and 4D indexing. The section following will discuss a series of increasingly difficult query categories on an RDF based MOB database. This taxonomy of queries is useful both to guide query implementation and in predicting performance.

AllegroGraph is both an RDF triple store and a quad graph store. It was designed to be hugely scalable (beyond core size). The four parts of each triple [sic] SPOG can hold any kind of data. All parts are efficiently linearly sortable. Along with string resources and literals, efficient specialized part encodings are supported: machine numerical types (fixed and float) as well as other specialized types. These encoded types generally sort in the natural order of the encoding.

Computer main memory and disk are both linearly addressable vectors. Computers are really good at doing things that are linear. It is well known that a vector of length n can be sorted in O(n log n) time and searched in O(log n) time. AllegroGraph is designed to exploit machine speed, despite scaling requirements, by keeping everything linear by maintaining multiple sorted indexes (e.g. SPOG, POSG, GOSP). By selecting the proper index, triples variously related to others can be retrieved from a local region of that index.

If a program wants to retrieve everything about <http://franz.com/employees#Jans>, all triples with this Subject are sorted together in the SPOG index. All triples with Jans as the Object are together in the OSPG index. All triples with a particular Predicate e.g. <http://franz.com/employees#isSupervisorOf> are grouped together in the POSG index, sorted secondarily on Object. Triples with particular values or one or more parts can be retrieved efficiently by a *cursor* that returns successive triples from one of the several indexes. Specialized *range cursors* return all triples over a range of values for a particular part and particular value(s) of other part(s).

Age, date and/or time, currency, phone numbers, stock prices, license-plate numbers, and barometric pressure are all linearly orderable. But Cartesian and spherical (e.g. geospatial) coordinates in two or higher dimensions are not immediately orderable and sortable. So the question is how to integrate these into the AllegroGraph linear indexing model? A particularly important problem is proximity search. We want to optimize speed retrieving all triples with coordinates in a certain locality for various kinds of localities: distance, bounding-box, and polygon.

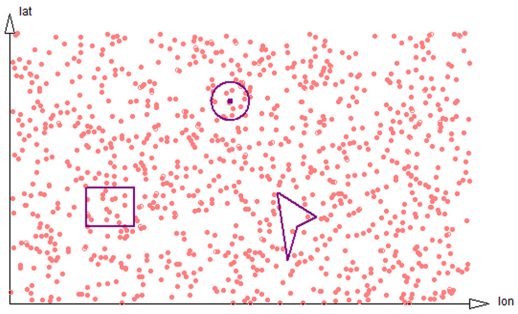

Fig. 1.

Data in two dimensions could be sorted on the two separate dimensions in the obvious way, either first on Y/latitude or X/longitude, or the reverse. But this causes search time to increase with the product of the size of the data set and the eventual number of data to be returned, i.e. O(n^2). As shown in the following diagram, the number of data points to be traversed increases with the product of the number of data points and the width of the region, or O(n^2) compared to the eventual number of data points to be returned.

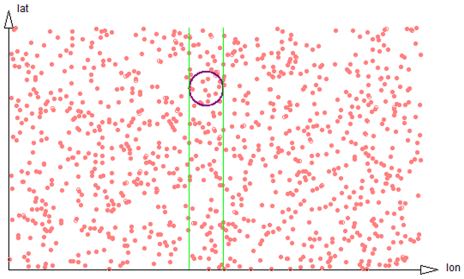

Fig. 2.

We'd much rather just search the region of interest, reasonably bounded in two dimensions instead of just one.

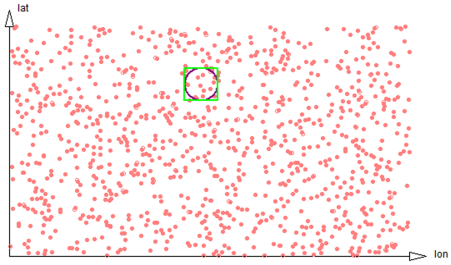

Fig. 3.

This yields scaling O(n), that is, a time proportional to the number of data points to be returned. In particular, if n is the number of points within the search radius, the number of points that must be considered is 4n/pi.

Within the requirements of AllegroGraph's fundamentally linear indexing, how can we avoid the unfortunate O(n^2) scaling? R-trees and various other indexing schemes support efficient search of two- and higher-order localities. But if all the data won't fit in memory, paging performance of R-trees can be unpredictable and data management can be convoluted. There is no obviously efficient way to reconcile 2-D and higher-order R-trees with AllegroGraph's linear indexing. We considered numerous fanciful ordering schemes, but devised nothing workable until we had the following breakthrough. Suppose we knew a little more about how we will use our data, specifically: The approximate size of typical regions to be searched. We could sort the data into strips.

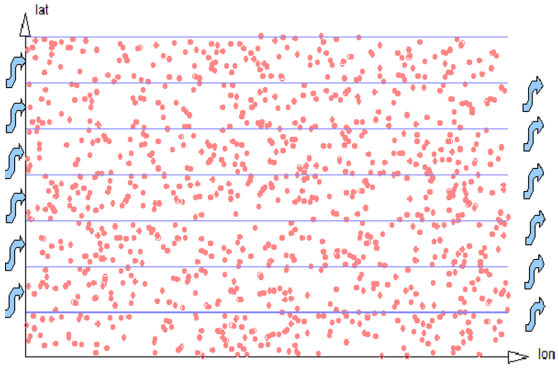

Fig. 4

In detail, a coordinate pair is converted to an unsigned integer. The major sort ordinate (latitude) is split into strip number and modulus within that strip. This requires mere integer division with remainder. (All this can be thought of as a variation on the technique of space-filling curves.)

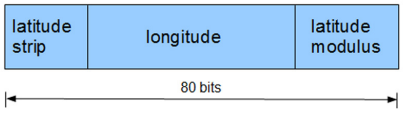

Fig. 5.

If the desired search diameter is exactly the same as the strip width, we need traverse short linear regions of just two strips. For a circular region, the number of data traversed is only 4/pi the size of the result set. AllegroGraph implements cursors as the mechanism for stepping through a range of data. A specialized class of cursor called a concatenated cursor can step though a set of linear segments of the data, e.g. the regions of the two strips below. It is assumed in these and all following examples

that triples would be used with the MOB identity as the Subject, a Predicate denoting a geo position, and the geo data encoded in the Object.

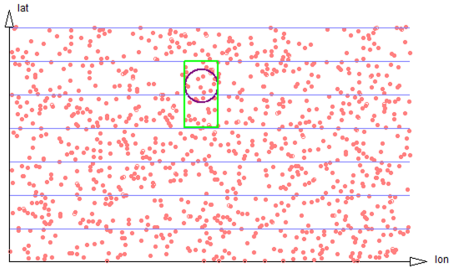

Fig. 6.

If the desired search diameter is smaller than the strip width, short regions of only one or two strips need be traversed. Efficiency stays high, falling off roughly linearly in the size of the over estimate in the expected search diameter.

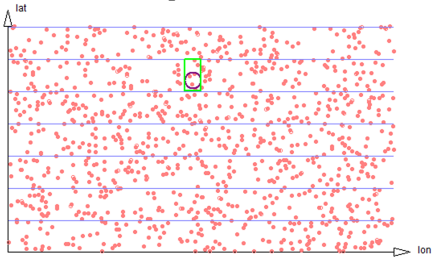

Fig. 7.

If the search diameter is somewhat larger than the strip width, the ratio between the number of data that need be traversed and the size of the result set stays fairly constant, but the number of separate linear strips that must be addressed and seeked increases about linearly with the error in estimate. As the number of separate linear regions that must be traversed increases there is of course an additional cost (the strip regions are not adjacent on disk) but performance stays reasonable even with fairly large estimation errors.

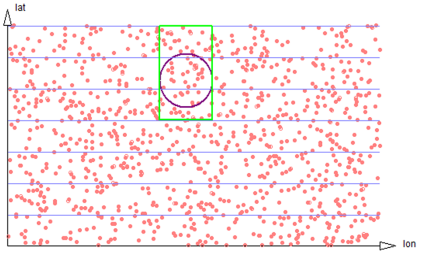

Fig. 8.

Finally, even if the search radius is much larger than the strip size, there is increasing cost due to the number of separate strip segments that must be traversed, but the proportion of the number of data that must be tested compared to the number of data in the result set remains close to 4/pi.

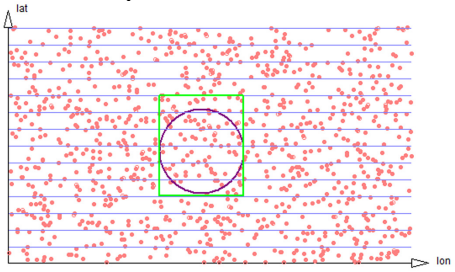

Fig. 9.

To summarize our 2D implementation, it may be inconvenient to need to specify strip width in advance. But performance is still reasonable even with an order of magnitude error. In addition, if extremely different strip sizes are needed for different purposes, the data can be stored twice with different strip sizes. A pedestrian wanting to locate a Starbucks has a different region of interest than routing software for the company trucks that deliver supplies.

The two coordinates in 2D can be anything, e.g. pressure and temperature or distance and time. This last possibility suggests extension to 3-D and beyond, particularly MOB data encoding paths in latitude/longitude/time.

Extending to 3D and 4D Geospatial

The 2D approach extends naturally to 3D and 4D, except data are sorted into 3D prisms instead of 2D strips. The prisms are sorted with time as the most-significant ordinate, since it will be particularly useful tracking a continuous path in time.

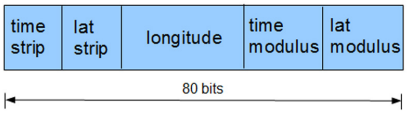

Fig. 10.

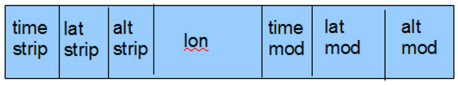

Fig. 11.

For a simple lon/lat/time proximity search, if the prism dimension is exactly the same as the search diameter, only short segments of four prisms must be traversed.

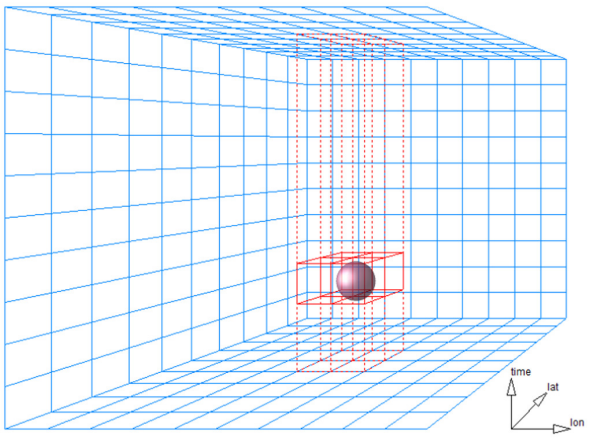

Fig. 12.

A bounding box in three dimensions would appear in the diagram above as a rectangular solid. The spherical region in the above diagram implies some scaling between time and distance. For a moving object this might be proportional to typical object velocity. Alternatively, the notion of proximity could be defined as any position within a given 2D radius and a particular time range; such a region would appear in the above diagram as a cylinder with a roughly vertical axis.

Recall that AllegroGraph queries employ cursors to traverse a range of consecutive triples within one of the linearly-sorted indexes maintained by the triple store. Many important queries on 3D (or 4D) data require following the track of a MOB through time, detecting other MOBs or fixed objects within a certain proximity of the first MOB. This is provided by a special kind of compound cursor that encapsulates a dynamically-varying set of individual cursors, one for each prism that overlaps the search region at any particular time point. As the compound cursor traverses through time (travelling upwards in the 3D diagrams of this paper) cursors for individual prisms are added or deleted from the compound as the search region overlaps or exits the individual prisms. The diagram below shows the prism regions considered at two specific time points while following a MOB single path.

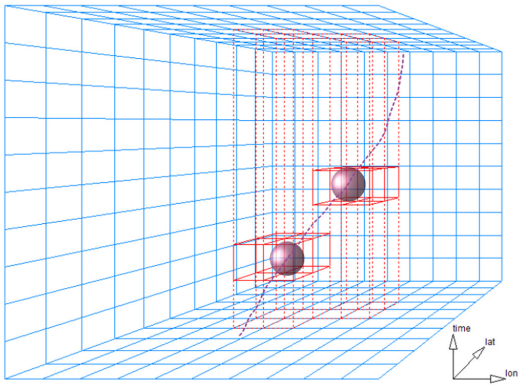

Fig. 13.

Several variations on this cursor are provided to facilitate the several kinds of query tasks outlined in the next section. It is critical that these compound cursors be implemented with care and with minimal overhead so that MOB queries can execute efficiently.

On the Difficulty of Various Categories of MOB Queries

This paper concludes with an informal illustration of five classes of MOB queries. These different kinds of queries have increasing degrees of difficulty, where difficulty is some combination of the complexity of the query along with the total portion of the data that must be traversed in performing the query. In judging difficulty, it is important to remember that happiness is finding things that are linear, as seen in this 1961 photo.

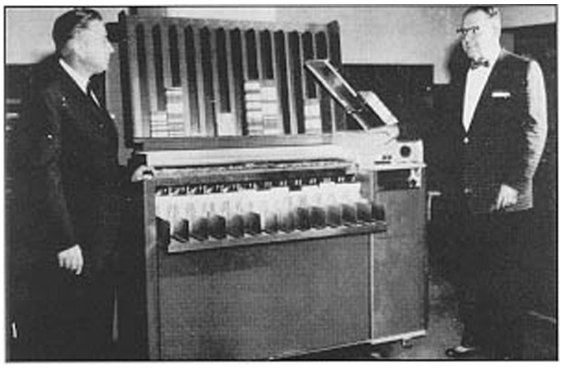

Fig. 14.

I. Simple proximity search that returns all 3D data with a certain proximity (or bounding box or other solid) around a given point.

This case has already been covered above, and the figure is repeated for completeness.

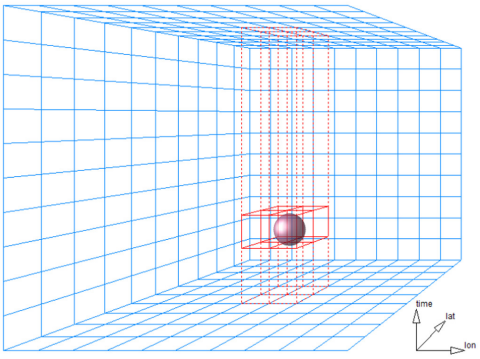

Fig. 15.

II. Given two particular MOB paths, determine if and when they were ever within a certain lat/lon/time distance.

This is best implemented by following the two tracks in parallel through time, repeatedly checking distance. The several prisms containing the regions around each track occur in local linear regions of the SPOG index. A special kind of geo path cursor is implemented which can traverse through time, adding or removing prisms as the path shifts in longitude and latitude. The two cursors need traverse only the portion of the data for the two given MOBs. All such data for each MOB is sorted together in time order in the SPOG index, minimizing the region of disk that might be paged.

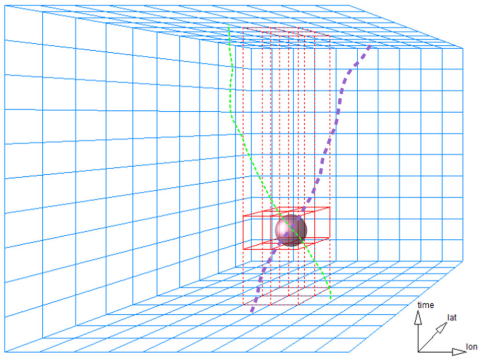

Fig. 16.

III. Given a single MOB, detect any other MOBs that ever come within a certain proximity.

This is somewhat similar to the previous, but the optimal implementation strategy is different. It is best to traverse the single MOB path to detect any other MOBs in proximity. Tracking along the MOB path in the SPOG index, those positions can be examined in the POSG index. Any data for other proximate MOBs will be in located nearby within the same several adjacent prisms in the POSG track being followed. Therefore, this search still requires only traversing the portion of the total data set along the single track of interest, although that path will be followed simultaneously in two different indexes.

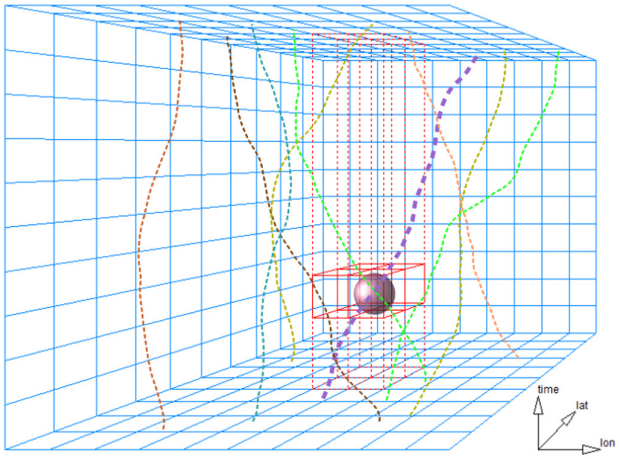

Fig. 17.

IV. Find all occurrences of any two MOBs within a certain proximity.

This clearly requires a traversal through the entirety of the MOB data using a different kind of cursor. However, since MOB positions within a given proximity must be within a local region proximate within one of several adjacent prisms, the scan can be done with a single traversal through the data, examining a running window on the POSG index. That region appears as a slab in the diagram below. Maintaining the moving slab places larger demands on memory than the previous query classes, but these demands are reasonable unless the slab in unreasonably thick.

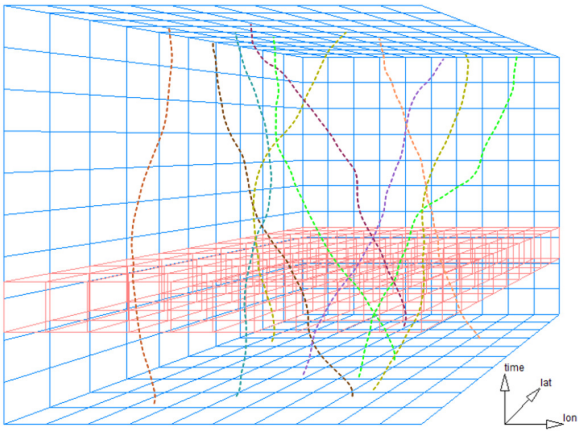

Fig. 18.

V. Detect potential Social Network Cliques between unknown MOBs, e.g. as evidenced by MOBs repeatedly being proximate in Place and Time, or being repeatedly suspiciously proximate in Place at different Times.

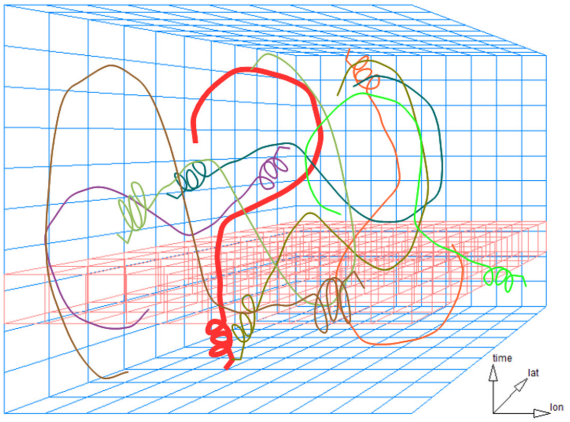

Fig. 19.

We don't know how to solve this because it isn't clear what we would be looking for, and it's difficult to find things when you don't know what you're looking for. Automatic detection of the unknown is an intriguing future research problem, hence (for now) the visual pun.