Where is the Beef?

Large-scale knowledge processing made possible only with AllegroCache

Wireless telecom companies have a serious problem. Growth of new mobile subscribers has slowed, and competing providers offer attractive packages and incentives to lure customers from one another. Consequently, customer churn rate is at an all time high, while companies struggle to increase Average Revenue Per User (ARPU). To reduce the churn and increase ARPU, one has to analyze the usage patterns and personal profiles of all customers and offer them compelling pricing/feature packages before they jump to competitors. But, this is easier said than done, considering that a major mobile provider can generate more than 100 million Call Detail Records (CDRs) per month in one state alone. Storing this enormous volume of data, easily reaching tens of terabytes over one year, is challenging enough. It is even harder to extract intelligence (such as calling patterns) from it.

The intelligence community faces an even more daunting task. To track down criminal elements, it needs to build a calling network (who calls whom calls whom, etc.) from the billions of CDRs, then search for specific calling patterns and do so in a timely manner. Traditional SQL databases are ineffective for such problems because calling networks do not naturally fit into relational tables. Furthermore, intelligence mining is more a graph-matching and pointer-chasing operation, rather different from SQL set operations. Attempting to do it with SQL databases requires continuous table joining and SQL calls, and renders the process too slow to be useful. One government agency resorts to mapping all the data (stored in billions of RDF triples) into the memory of a super computer with many terabytes of RAM before performing analyses. Simply rebooting the system and reading the data into memory takes one week. But how many of us can afford such luxurious equipment?

BILLIONS OF RDF TRIPLES

The call network is just one type of semi-structured information, unsuitable for relational databases. By most estimates, more than 85% of corporate data is unstructured (e.g., Word documents, HTML pages, etc.), not residing in any database. Therein lies most corporate knowledge, unintelligible without human interpretation. Simply converting such data into XML documents and storing them in databases does little to enhance knowledge understanding and its application.

One emerging solution is to transform such data into RDF triples (**s**ubject, **p**redicate and **o**bject). RDF stands for Resource Description Framework from the W3C Semantic Web working group, and is implemented in XML. It borrows heavily from early research in knowledge representation and management. The flexibility of RDF makes it suitable for representing any knowledge (be it CDR networks or contact networks). **However, as shown previously, the number of RDF triples can easily grow into millions or even billions for real-world applications, making it difficult to process efficiently with traditional means. Small wonder there is scarcely any deployment of practical RDF applications.**

Relational databases are inefficient for storing and navigating RDF triples. Object databases, while better able to model RDF triples, still scale poorly when the number of triples exceeds tens of millions. We need a tool with very effective indexing mechanisms and a high-speed Btree (or B+tree) to store, retrieve and navigate huge numbers of triples.

HIGH-PERFORMANCE TRIPLE STORE

Franz has developed **AllegroGraph** specifically for managing very large-scale RDF triples. It employs 8 indices, each is stored on disk with either a B+tree or an AVL tree. The low-level API consists of only: **store-triple(s,p,o), get-triple([s],[p],[o])** and **read-file**(uri). It includes an expressive query language, **RDF Prolog,** particularly suited for graph search and graph matching over an RDF network. It can find semantic relations between RDF nodes automatically, using complex Prolog clauses as needed without speed degradation. For example, the following RDF Prolog statement defines a father relation on top of AllegroGraph:

```
(<-- (father ?x ?y) (male ?x) (q ?x !o:has-child ?y) )
```

To test its scalability, two available large data sets, the Lehigh University Benchmark and a movie and

actor database, were used. Using an AMD 64, 2 Ghz, CPU machine with 16 GB of RAM, the burst rates of parsing RDF and storing triples are:

Size of Data Set	< 200 million	> 200 million
# of Triples / Sec	10,000	4,000

This performance is almost an order of magnitude faster than known non-memory-resident benchmark results. Retrieving RDF triples is just as fast. On average, the retrieval times are:

	Btree Miss	Btree Hit	Query Cache
Time / Triple	30 x 10^{-6}s	1.8 x 10^{-6}s	0.3 x 10^{-6}s

This remarkable result is achieved with 8 indices, which adds a disk storage overhead of about 300 bytes per triple but offers unparalleled performance.

ALLEGROCACHE – OODB TOOL KIT

All the functions used to create AllegroGraph (such as user-defined serialization, Hash Tables, B+trees, AVL trees, etc.) are all part of an object database, AllegroCache. Rather than offering a monolithic database system (like most commercial offerings), AllegroCache makes its major functions available as self-contained tools with simple APIs for programmers to customize the database system for their special needs. For example, storing and processing a monotonic data set (such as stock transaction data and most knowledge data) does not need full transaction capability and its attendant overhead. A B+tree is more efficient to use. On the other hand, when full transaction integrity is required, AllegroCache offers full **ACID** compliant transactions with object-level locking. To test its scalability, a billion objects (each with 3 attributes) were created, then randomly accessed. Test #3 involves a schema update (described later). Overall, the performance of AllegroCache is quite satisfactory.

Test	Index	# of Classes	Object Creation Time	Random Object Access Time
1	No	1	17.6 K objects / sec	
2	Yes	3	**10K objects / sec**	**13K objects / sec**
3	Yes	3	**(Update Schema)**	**9.7K objects / sec**

ZERO IMPEDANCE MISMATCH

One of the hairier aspects of modern application programming is the impedance mismatch between programming languages and database access languages (mainly SQL and its derivatives), both in language syntax and semantics and in data types. This necessitates constant mapping of one data type to another and switching language semantics within an application, inevitably leading to errors and frustration.

Object databases go a long way to mitigate such impedance mismatch, allowing programmers to focus more on object persistence rather than on object decomposition into rows and columns. AllegroCache is by far the most seamlessly integrated, with its database query language the same as its Lisp programming language. For example, the code below creates a PhonebookEntry database:

```
(open-file-database "Phonebook"
                    :if-does-not-exist :create
                    :if-exists :supersede)

(defclass PhonebookEntry ()
  ((FirstName :initarg :FirstName :index)
   (LastName :initarg :LastName :index)
   (PhoneNumber :initarg :PhoneNumber :index)
   (PostalCode :initarg :PostalCode :index))
  (:metaclass persistent-class))

(make-instance 'PhonebookEntry :FirstName "John"
  :LastName "Doe" :PhoneNumber "510-452-2000" :PostalCode 94607)

(retrieve-from-index 'PhonebookEntry 'PostalCode 94607)
(doclass (obj 'PhonebookEntry) (print obj))

(commit)
(close-database *allegrocache*)
```

This code is no different from coding against in-memory data, except a few database-specific statements like **open-file-database, commit,** and **close-database**.

DYNAMIC SCHEMA UPDATE

Changing a database schema is usually a very laborious and time-consuming process, not to be attempted casually. AllegroCache, on the other hand, allows programmers to change database schema at runtime.

```
(open-file-database "Phonebook")

(defclass PhonebookEntry ()
  ((FirstName :initarg :FirstName :index)
   (LastName :initarg :LastName :index)
   (PhoneNumber :initarg :PhoneNumber :index)
   (PostalCode :initarg :PostalCode :index))
   (Member :initarg :Member)
  (:metaclass persistent-class))

(doclass (obj 'PhonebookEntry)
  (setf (Member obj)
    (retrieve-from-index 'PhonebookEntry
      'PhoneNumber (PhoneNumber obj) :oid t)))

(commit)
(close-database *allegrocache*)
```

Here, an existing object class, PhonebookEntry, is redefined with an added attribute, Member, to store all persons sharing the same phone number. Object instances that were created and stored with the old schema are automatically and quickly updated to the new class definition when they are accessed. The ability to update the schema at runtime is tremendously useful during application development when the object and database design undergoes constant evolution. It is also very desirable when dealing with domains, such as Bioinformatics, that are ambiguous and continuously change.

For more information on AllegroCache, see http://www.franz.com/products/allegrocache, or contact **Steve Sears** at +1 (510) 452-2000 x 154.