555 12th Street, Suite 1450
Oakland, CA 94607
510-452-2000
www.franz.com

AllegroCache™

AllegroCache — Persistent AI Built-in, All the Way Down™

Table of Contents

Complex Applications Are Pervasive Today 3
What Is AllegroCache 5
AllegroCache System Architecture 5
AllegroCache Extension to Lisp 7
AllegroCache as an Intelligent Database System 9
AllegroCache Performance Benchmarks 10
When to use AllegroCache 12
Initial Validating Applications 13
Platforms Supported 15
Product Configuration 15
Summary 15

AllegroCache — AI Built-in, All the way down™

AllegroCache™ is a high-performance, dynamic object caching database system. It allows programmers to work directly with objects as if they were in memory while in fact the object data is always stored persistently. This greatly simplifies programming tasks, enabling programmers to focus on solving the complex problems at hand. It supports a full transaction model with long and short transactions, and meets the classic ACID requirements for a reliable and robust database. It automatically maintains referential integrity of complex data objects. Furthermore, AllegroCache™ provides 64-bit real-time data caching on the application memory to achieve very high data access throughput by applications over as large a data set as necessary.

In this white paper, we will first articulate the growing needs for such an object database, then describe technical characteristics of AllegroCache™ especially from the viewpoints of Lisp applications, discuss when and how to employ AllegroCache™ for complex intelligent software applications, and finally highlight a few initial applications using AllegroCache™.

Complex Applications Are Pervasive Today

The problems we are facing today are of unprecedented complexity and scale. In the commercial sector, businesses need to deal with the accelerated pace of changes brought on by the Internet. E-commerce sites need to provide context-relevant solutions to millions of customers each with their own preferences and purchasing habits. Banks must constantly monitor millions of global financial transactions to detect financial fraud and money laundering. In the government sector, agencies fighting terrorism must deal with an explosion of intelligence and event alerts, and the signal-to-noise ratio is getting worse as more data is collected. In the scientific community the problems are just as daunting – consider Bioinformatics, drug discovery, GIS, and Genealogy. Complexity is no longer the realm of scientific research but pervade every sector today. (See figure 1)

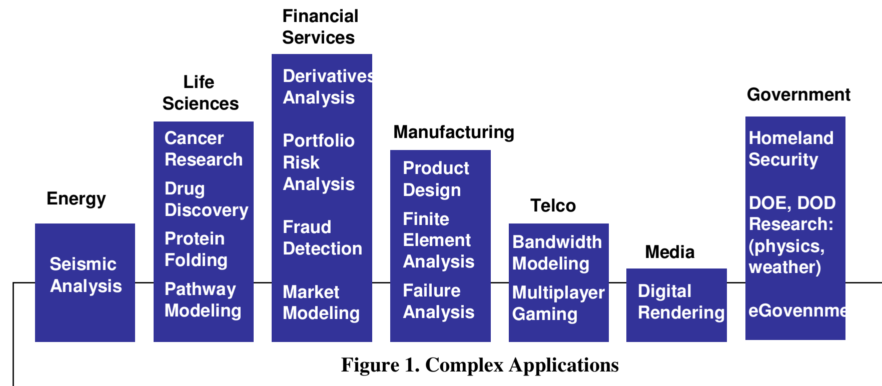


Figure 1. Complex Applications

Data from J.J.Porta of IBM, 2003

For example, in the field of life science and medicine, researchers have long recognized the significant clinical heterogeneity among tumors, even though they all look similar using standard pathology tools. With Microarrays, those tumor tissues previously considered similar now clearly exhibit distinct expression patterns, thus rendering systematic tumor classification finally possible. (See figure 2)

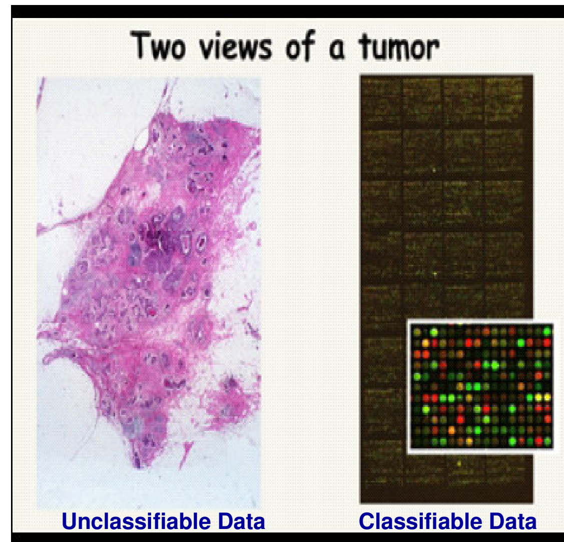


Figure 2

However, to understand the biology and the disease process behind each expression pattern in order to devise an effective medical treatment requires correlating it with information and knowledge from many other data sources (such as gene ontology, protein databases, metabolic pathway databases, etc.) As a result, a great promise ends up creating an even greater computational challenge.

Such is not unique to the life science field. While relational database and the SQL query language have in the last 20 years revolutionized corporate data processing and will continue to play their key enterprise roles, they no longer suffice to handle today's complexity. This is so because, in most complex cases, the data contains deeply nested recursive structures and active data slots that cannot be easily fit into a relational data model. Similarly, applications trying to reason over such data sets frequently need to perform recursive queries on them. Imagine queries like the one below to search for the origin of a suspicious financial transaction shrouded in a convoluted transaction web:

```
"Transaction-origin (X,Y) :- Transaction-from (X,Y);
       Transaction-from(X,Z), Transaction-origin (Z,Y)."
```

It is hard to do this query straightforwardly in SQL. Further complicating the situation is the constant need to cross-reference over very large amounts of data. We need an object model with in-memory caching on the application level to supplement existing relational databases if progress is to be achieved within a reasonable time frame.

What Is AllegroCache

AllegroCache™ is a high-performance, persistent, dynamic object caching system. It allows programmers to work directly with objects as if they were in memory while in fact the object data is always stored persistently. It supports a full transaction model with long and short transactions, and meets the classic ACID requirements for a database. It maintains referential integrity of complex data objects. For example, a pointer to a deleted object is automatically and silently changed to nil, which is essential for frame-based knowledge representations with an extensive pointer network. Furthermore, AllegroCache™ provides 64-bit real-time data caching on the application memory to achieve very high data access throughput by applications over as large a data set as necessary. The table below compares AllegroCache™ to other currently popular database technologies.

	RDB	OODB	AllegroCache
Application Level Caching		√	√
Dynamic Object Schema Evolution			√
Garbage-collected, Multi-threaded, Distributed			√
AI Built-in, all the way down™			√

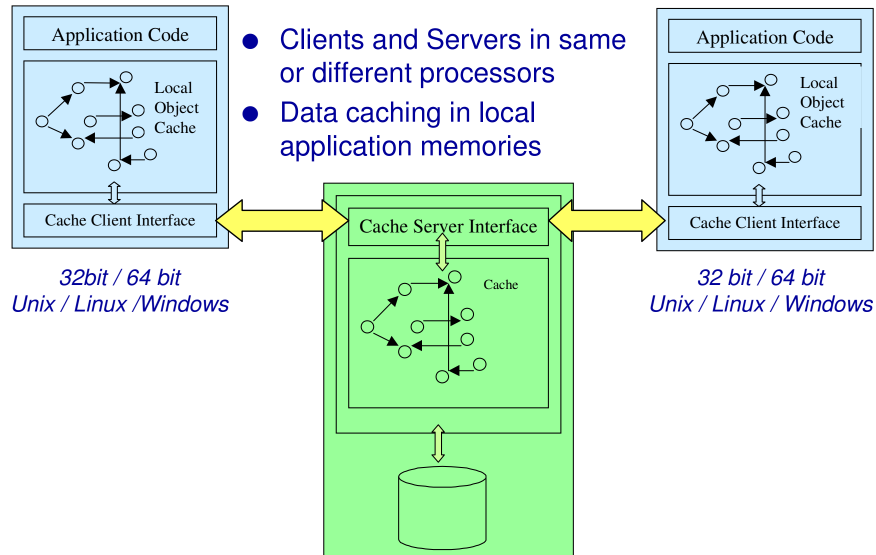
AllegroCache runs on Windows, Unix and Linux for both 32-bit and 64-bit architectures. It works fine on traditional 32 bit Windows and Unix system but really shines on multiprocessor 64 bit systems with large memories.

With respect to its application, we can look at AllegroCache in two ways. We can view AllegroCache as a high-performance, application-level caching system with persistent data store. Alternatively, we can look at AllegroCache as a natural extension to Common Lisp. From such a perspective, AllegroCache is a powerful, persistent object-oriented layer on top of the Common Lisp Object System (CLOS). Effectively, AllegroCache turns Lisp into a database and database language for tomorrow.

AllegroCache System Architecture

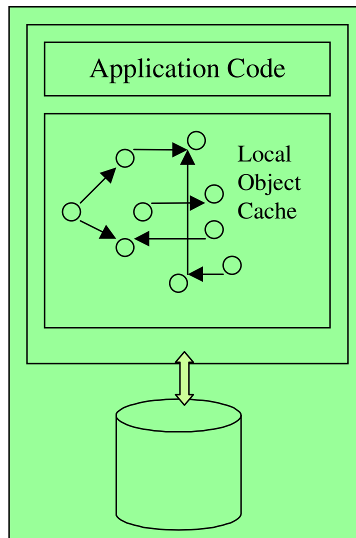
AllegroCache provides both a single user and multi-user environment. In the multi-user environment, different users (or processes) on different processors can access the same database over the network.

A typical configuration with multiple users would be:



This picture shows multiple machines talking to one machine that holds a large cache. The two machines at the top are clients that retrieve their objects through a cache-client interface.

You can also use AllegroCache in single-user mode. In this case the application, the cache and the b-tree interface all reside in one process.



One of AllegroCache's unique programming features for Lisp application developers is that *the Lisp programming language is the database query and manipulation language*. In essence, with AllegroCache you program your application with persistent data as if it were in memory without having to

marshal data in and out of the database. The machinery behind AllegroCache will automatically ensure such data persistence occurs.

This is significant because it resolves one of the hairier aspects of modern application programming — the impedance mismatch between programming languages and the database access languages (mainly SQL and its derivatives). This mismatch occurred because accessing relational databases became common in applications *after* programming languages had been standardized and were in wide use. Therefore, almost all modern programming languages use separate database access libraries for accessing RDBMS data within applications. Not only does the syntax and semantics of the two languages likely differ (sometimes subtly), native data types within a programming language are often incompatible with the database data types. The need to constantly map one data type to another and switch language semantics within a single application is inevitably "error-prone and frustrating."[1]

Lisp, being a functional language with extremely simple syntax, is able to extend itself (with new Lisp macros to handle database transactions) to be the *native* database access language without any impedance mismatch. Whatever data mapping (marshalling) is necessary is totally hidden from programmers using the standard CLOS class mechanism with a special new metaclass. AllegroCache with the Lisp programming interface essentially alleviates from programmers the task of having to program special code for persistent data while writing applications.

This is akin to the advent of virtual memory (VM) in the 70's that relieved programmers the headache of having to overlay application code to cope with the limited RAM memory space.

AllegroCache Extension to Lisp

As part of AllegroCache, small syntactical extensions to Allegro CL and an AI query language are added. These extensions blend seamlessly to the Lisp language. Some specific programming interfaces in Lisp are described below.

(1) **Metaclass: database-class**

By using this new database-class metaclass in an object class definition, all its instances are automatically stored on disk in a database. We call these instances persistent instances. The power of this concept is that you as the Lisp programmers can work with CLOS objects as if they were all in memory.

```
(defclass person ()
  ((name)
   (address))
  (:metaclass persistent-class))
```

(2) **Unique object identifiers**

Every instance created for a class of database-class will have an object identifier (*oid*) which is guaranteed to be unique and will never change during the lifetime of the database. Programmers will have access to this identifier so that objects can be retrieved or deleted based on this *oid*.

[1] See Joel on Software http://www.joelonsoftware.com/items/2004/03/25.html

(3) **Referential Integrity**

In many cases the data a program operates on is a complex graph or a pointer network, i.e., instances pointing to other instances. When objects are deleted, all references to those deleted objects will be automatically changed to the *nil* value.

(4) **Maps and Indexes**

When working with a huge number of objects, you need indexing mechanisms to enable you to quickly find a particular object. Therefore, AllegroCache introduces the concept of a *Map*. A *Map* is a set of key-value pair where the keys and values can be almost any Lisp type. Maps can either live in memory only or be stored in the database. A Map can contains 1-1, many-1, 1-many, or many-many relations. A Map where the value is a persistent object in the database is called an Index. When defining a class, you can specify which slots should be indexed.

```
(defclass person ()
  ((name)
   (age :index :any)
   (social-security :index :any-unique)
   (income)
   (father)
   (city))
  (:metaclass persistent-class))

(defclass city ()
  ((name :index name)
   (state)
   (country))
 (:metaclass persistent-class))
```

(5) **Dynamic Class Schema Update**

One of the unique features of AllegroCache is that you can redefine classes on the fly at runtime. When your application redefines class definitions at runtime, AllegroCache will automatically and lazily change existing instances when and only when the application accesses them. And, yes, every instance has a class-version number.

(6) **Lisp as the Retrieval Language**

In principle you don't need a specialized retrieval language to work with persistent CLOS instances. As described previously, Lisp programmers view the database as if all the instances are in memory. Therefore, you can write your programs to operate on persistent CLOS instances as you will do without using AllegroCache. AllegroCache, however, does provide tools that let you loop over every instance of a particular persistent class.

(7) **Prolog as a Retrieval Language**

In some cases when you work with deeply nested structures and when it becomes too hard to remember which slots are indexed, you might want to use a more declarative way to express your search or graph-matching algorithms. Because AllegroCache is an orthogonal feature to the rest of Lisp, reasoning programs that work on CLOS objects will seamlessly

work with AllegroCache. For example: you can easily use the built-in Prolog in ACL to reason over a set of CLOS objects. Certain changes are made to ACL Prolog so that it works better with AllegroCache. First, some syntactic sugar is added so that queries over CLOS objects read much better, and Prolog is made aware of indexes that might exist for certain slots.

AllegroCache as an Intelligent Database System

Besides being a high-performance persistent object layer for CLOS, AllegroCache is also a full-fledged database. That is, AllegroCache meets all the production requirements of a traditional Relational Database Management System (RDBMS).

(1) **ACID Test**

Every production database management system needs to achieve four goals: atomicity, consistency, isolation and durability (ACID). Databases that fail to meet any of these four goals is not considered reliable. AllegroCache meets all these essential ACID requirements.

- Atomicity states that database modifications must follow an all or nothing rule. Each transaction is said to be atomic, if one part of the transaction fails then the entire transaction fails.

- Consistency states that only valid data will be written to the database. If, for some reason, a transaction violates such databases consistency rules, the entire transaction will be rolled back and the database will be restored to a prior state that is consistent with those rules.

- Isolation requires that multiple transactions occurring at the same time not impact each other's execution. For example, if one issues a transaction against a database while another issues a different transaction on the same database, both transactions should operate in an isolated manner. Note that the isolation property does not ensure the order of transaction, merely that they will not interfere with each other.

- Durability ensures that any transaction committed to the database will not be lost. Durability is accomplished through database backups and using transaction logs to restore committed transactions in spite of any subsequent software or hardware failures.

(2) **Transactional Model**

AllegroCache supports a full transactional model. You can perform long and short transactions, commits and rollbacks, and the database supports complete recovery if the machine should ever fail during the commit.

(3) **Concurrency**

In a multi-user environment, there are two models for updating data in a database: optimistic concurrency, and pessimistic concurrency. In general, AllegroCache will work with the optimistic model for concurrency. More and more modern databases support the optimist form to gain better database performance. For example in SQL Server, Microsoft promotes the so-called DataSet object which is designed to encourage the use of optimistic concurrency for long-running activities such as accessing data remotely while other users are interacting with the data. An excerpt from Microsoft on this subject is included below for your reference.

From: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconoptimisticconcurrency.asp

Pessimistic concurrency involves locking rows at the data source to prevent users from modifying data in a way that affects other users. In a pessimistic model, when a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the lock owner releases it. This model is primarily used in environments where there is heavy contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

Therefore, in a pessimistic currency model, a user who reads a row with the intention of changing it establishes a lock. Until the user has finished the update and released the lock, no one else can change that row. For this reason, pessimistic concurrency is best implemented when lock times will be short, as in programmatic processing of records. Pessimistic concurrency is not a scalable option when users are interacting with data, causing records to be locked for relatively large periods of time.

By contrast, users who use optimistic concurrency do not lock a row when reading it. When a user wants to update a row, the application must determine whether another user has changed the row since it was read. Optimistic concurrency is generally used in environments with a low contention for data. This improves performance as no locking of records is required, and locking of records requires additional server resources. Also, in order to maintain record locks, a persistent connection to the database server is required. Because this is not the case in an optimistic concurrency model, connections to the server are free to serve a larger number of clients in less time.

In an optimistic concurrency model, a violation is considered to have occurred if, after a user receives a value from the database, another user modifies the value before the first user has attempted to modify it.

(ref: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconoptimisticconcurrency.asp)

(4) **cache consistency**

Cache consistency overlaps with the concurrency issue. You (or your application) will always see a consistent version of the database, although your own private version may be slightly out of date (which will be synchronized when you try to commit the transaction).

AllegroCache Performance Benchmarks

Besides functionality, scalability and performance are two critical factors in any production database. The following benchmarks are intended to provide an objective measures of AllegroCache in these two areas.

For scalability, we conducted three benchmarks to test its performance characteristics with huge databases. The results look very promising.

- **Test #1**: We created 1 billion objects of the same object class in the AllegroCache database without indexing. The object creation time was more or less constant, independent of the size of the database.

- **Test #2**: We created 600 million interconnecting objects (objects pointing to one another) of 3 different object classes in the AllegroCache database with one of the slots indexed. Again, the object creation speed was more or less constant throughout the test, albeit slightly slower than test #1. We then accessed 1 million objects randomly using the indices, then retrieved the slot values of each object. We closed the database at the end of the test.
- **Test #3**: We opened the database created in test #2 for update, then redefined one of the classes (by adding one additional slot) at runtime. We then randomly accessed 1 million objects of the changed class. As each object was fetched into memory, it was updated automatically to include the new slot. A value was then written into the new slot.

All of the tests were conducted on a 64-bit AMD machine with more than 2GB of RAM. The results are summarized below.

AllegroCache Benchmark

Test	Indexing	# of Classes	# of Objects	Object Creation Time	Random Object Access Time	Process Size	Databse Size
1	No	1	1 billion	17.6 K objects / sec		260 MB	94 GB
2	Yes	3	600 million	**10K objects / sec**	**13K objects / sec**	2 GB	65 GB
3	Yes	3	600 million		**9.7K objects / sec**	2GB	65 GB

The above benchmarks demonstrated that AllegroCache could:

- Manage more than 1 billion objects reliably and with ease, even redefining the class definition dynamically at runtime
- Manage and navigate highly associative object models with good speed
- Support ACID compliant transactions without undue performance penalty

We further benchmarked the performance of AllegroCache against a widely used and commercially deployed relational database, MySQL. The benchmark consists of creating 5 million CDR (Call Detail Record) objects, reading 1 million objects sequentially (from the database on disk without pre-caching) and reading 1 million objects randomly (through the primary-key indices and again without pre-caching). All tests were conducted on the same 64-bit AMD machine. Detailed results are presented in the table below.

Again, AllegroCache fared very well against MySQL. For object (record) creation, AllegroCache and MySQL performed at similar speed, about 6,000 objects (records) per second. Please note that the object class in this benchmark was much more complex than that in the above scalability benchmarks, thus the slower object creation speed. For sequential record (object) read, MySQL performed twice as fast as AllegroCache. This is to be expected since relational database is tuned to handle such a task. However, for random object (record) read, AllegroCache performed 7 times faster than MySQL, a dramatic speed-up over a popular relational database. Speed of random read is very important when doing graph search and object pointer chasing, for random read is typical in dealing with complex, highly associative or network data models.

Performance Comparison vs. Relational DB (MySQL)

	AllegroCache	MySQL
Object Data (Record) Definition	(defclass* call-data () ((call_number :index :any-unique) action from_user to_user time_start time_spoken amount balance description))	create table call-data (call_number int primary key auto_increment, action int, from_user int, to_user int, time_start int, time_spoken int, amount int, balance int, description varchar(200));
Write	5.5K Objects / Sec	6K Objects / Sec
Sequential Read	35K Objects / Sec	70K Objects / Sec
Random Read	35K Objects / Sec	5K Objects / Sec

When to use AllegroCache

When should you use AllegroCache instead of a traditional RDB? The following examples highlight some of these criteria.

(1) Your needs cannot be handled with SQL. For storage purposes your data is in a relational database. However, your data actually forms a complex graph of objects pointing to other objects. So to do any serious query on your data, you really need to employ graph-matching algorithms or recursive queries. This is simply too painful to do it straightforwardly in SQL. A typical example is a genealogy database where you want to find a common ancestor of two random persons. This is basically a search problem and not a flat SQL retrieval. Using languages such as Prolog (built-in within ACL) for such queries is both natural and more efficient.

```
"Ancestor (X,Y) :- Parent (X,Y); Parent(X,Z), Ancestor(Z,Y)"
```

Another example is the Stanford BioBike Bioinformatics application to be described later.

(2) Your problem can be solved by SQL but you want to bypass going through SQL every time you need an object from the database. One example is RDF databases. You may be able to store RDF triples in a traditional relational database, but reasoning over the data proves to be very slow. Custom-made databases for RDF are much faster. You can easily make a customized RDF database with AllegroCache. All subsequent queries on the RDF data will only involve referencing the data in memory directly.

(3) Your data is too complex for a relational database. For instance, your objects are frames with variable slots, and new slots are to be added at run-time. In such a case, it would be better to use a frame system on top

of AllegroCache rather than a relational database. With AllegroCache, you can even index certain slots at runtime and the retrieval language immediately recognizes them.

(4) Use AllegroCache to improve performance in database backed e-commerce sites. As the Internet matures, accessing information and services through the web at any time from anywhere by anyone has become necessity for business to survive. B2C e-commerce sites like Amazon.com are prominent examples of such a new business model. The explosive growth of Internet-based business services means that thousands may try to access your application and data simultaneously, anytime. To succeed, applications must avoid poor response time and service outages. Simply adding bigger hardware and more relational database systems (e.g., Oracle servers) adds significant costs without addressing the scalability problem, especially for accessing non-static data. This is due to logical database access bottlenecks even though bottlenecks at physical database servers may have been reduced. This is really not an intelligent solution since, at typical e-commerce sites, 90% of transactions are reads and 90% of queries access only 10 % of the data. It is more sensible to deploy a 64-bit multi-gigabyte machine that caches the data in the form of objects. (a 16-Gigabyte server with four AMD processors costs at the time of this writing not much more than $14,000). Amazon has used this application-level caching approach quite successfully, reducing some of data synchronization delay from more than 24 hours to minutes. This is one type of application that originally motivated the development of AllegroCache.

Initial Validating Applications

While developing AllegroCache, we have actively cooperated with several complex application projects (see below) at outside institutions and at Franz to validate the design of AllegroCache and to fine-tune performance characteristics. We want to ensure that AllegroCache meets the strict requirements of these complex applications, and that it will be production-ready when launched. Here are some examples:

(1) **BioBike at Stanford University**

The BioBike project provides an integrated, on-line, programmable biological knowledge base for the biologists. BioBike itself is a very simple, very efficient biology-specific programming language. The BioBike vision is having this programmable access to an integrated biological knowledge base, through the web. In the best case, biologists themselves would be able to log into the system and write their own simple programs to ask novel biological questions.

(See http://nostoc.stanford.edu/Docs/intro.html)

In cooperation with the authors of BioBike, we have implemented a version of BioBike where the knowledge base is stored in AllegroCache. BioBike uses a frame-based knowledge representation thus requiring us to build a frame-based system on top of AllegroCache. When using frames with AllegroCache, you can add arbitrary slots to a frame object, similar to the Python's object system.

With this added capability, AllegroCache can now read in heterogeneous databases without worrying about the number of object slots to be read in.

(2) **Fraud Detection over Call Detail Records (CDR's)**

Kido Software in India is testing Allegro Cache for building mobile user profiles from call data records (CDRs) collected from downstream processes. Mobile phone operators use such profile information to analyze fraudulent uses of mobile phones. This profile information is further used for analyzing detailed caller use patterns, which are useful in planning and efficient deployment of network resources.

Mobile operators heretofore are struggling with the volume of call data that typically run into millions of calls per day/ week/ month, depending on the size of the operator. This results in gigabytes of call data in short order that need to be analyzed in a timely manner, ideally in near real time.

Allegro Cache is a very good database choice for such application because of its scalability and its in-memory cache for high performance. It has proven effective for large amounts of CDR data running into hundreds of gigabytes.

(3) **Textual Data-mining**

Franz Inc runs the largest chat-bot site on the web (see www.pandorabots.com). There are about 50,000 bots hosted on the Pandorabots site, and daily conversation has reached over 1 million sentences. It has stored more than 100 million conversations in AllegroCache (a conversation is roughly defined as the conversation id, the IP-address, the user id, the sequence number, what the user typed, what the bot answered, the topic of the sentence, etc.) With these conversations cached in AllegroCache, we can do interesting graph-searches through the whole conversation space using full indexing on all the slots of these 100 million conversations. It is used to validate both the performance and scalability of AllegroCache.

(4) **RDF knowledge server**

Another application written with AllegroCache is a RDF knowledge server. RDF is the language of the Semantic Web for representing information about resources in the World Wide Web. It was originally intended for representing metadata about Web resources, such as the title, author, and modification date of a Web page, copyright and licensing information about a Web document, or the availability schedule for a shared resource. By generalizing the concept of a "Web resource", RDF can also be used to represent information about things that can be identified on the Web, even when they cannot be directly retrieved on the Web. Examples include information about items available from on-line shopping facilities (e.g., information about specifications, prices, and availability), or the description of a user's preferences for information delivery. (See http://www.w3c.org/TR/rdf-primer/#intro.)

Franz has implemented a RDF knowledge server that stores triples and nodes (URI's that denote subjects and predicates) in AllegroCache. The query engine uses a number of indexes on triples and nodes to retrieve triples very efficiently. It uses the ACL XML parser to read in existing RDF files on the web and use AllegroServe and Webactions to serve the web pages for users.

Platforms Supported

AllegroCache runs on all 32 and 64 bit platforms where Allegro Common Lisp runs, including basically all Windows, Unix, Linux and Mac OSX platforms.

Product Configuration

Configuration:

(1) Single machine, multiple processes simultaneously accessing the database on that machine.

(2) Multiple processes on multiple machines running different OS access a database on a single sever machine in the network simultaneously.

Developers version: For developers in the government or in commercial sectors. All customers of ACL Enterprise Edition automatically get a copy of AllegroCache developer version for their evaluation and development needs.

Evaluation version: Non-ACL customers can contact Franz to get a 6-month evaluation of both ACL and AllegroCache from Franz (to be extended if necessary).

Deployment version: We have three deployment options:

32 bit single-user version (stand-alone version)

64 bit one processor server version (server application backend)

64 bit multiprocessor server version (server application backend)

Please call 1-888-CLOS-NOW or email to:

`allegrocache@franz.com`

Summary

AllegroCache is the database for tomorrow

- Computational complexity of problems we faced today are growing exponentially, and the situation will only worsen.
- Distributed object data caching on application memories offers the best performance when dealing complex data model.
- AllegroCache is the easiest and most versatile data caching system for handling complex problems with large data sets.
- AllegroCache has AI built-in for easy development of reasoning code over large complex data sets.