Peter Joh – 8/07/2013
Document V0.01

Future of Hierarchy
Distributed Persistence using Ram-Based Persistence
Requirements

Overview:

This is the future vision of Frictionless Persistence. Persistent matrices will be able to be distributed across multiple servers with the goal of near zero-latency of the synchronization of shared data across the server instances. A distributed application would be able run at speeds near those as if all the applications instances, services, and DB instances were running on the same server. The approach is actually not new, and is also a trend that has been developing for years. But it's an approach that is ideal for Hierarchy persistence.

It's both a hardware and software solution where the server instances pass data-sync messages on an ultra-fast message bus that hooks as directly into the cpu's as possible. This bus is used for short messages to notify other cpu's of short read and write sync messages (it is the same idea that hardware caches use to sync data across multi-processor systems). This bus is also used by the server instances to pass the actual values for small writes-operations. The usage of this ultra-fast bus allows the distributed servers to synchronize their data at speeds in the ballpark of the cpu's clock, which means as a whole, the distributed system runs nearly at the speed of the cpu.

The disadvantage of this approach is that there is typically a limitation as to the number of CPU's the system as a whole can have, as the number of synchronization messages can become overwhelming as the number of processors increases (we've been told this limit is around 100 to 200 processors). But, the advantage is that if your application is running at near CPU speeds for most operations (meaning with almost no network-latency for communication between nodes in the cluster), you typically can run enormous volumes of requests with relatively few processors!

NOTE: This document is initial research and needs much further development before it can be considered a valid approach! (and may just be an impractical pipe dream). But, we wanted to at least present this goal for the future of Frictionless Persistence, giving us all something to shoot for.

Goals of RAM-based persistence:

- To allow devs to work as naturally as possible with persistence data. More specifically, one of the most natural ways for devs work with persistence is to have the native data-structures, collections, and objects within the programming language directly support persistence.

- To create a distributed persistence solution that performs nearly at the same speed as running a non-persistence solution!

  Specific Targets:
  - Inter-server synchronization messages to occur in one read/write cycle.
  - On each server, the Memory-Status Directory responds in 1 cycle (actually less).
  - Writes to non-volatile memory happen in less than 1 read/write cycle (using DRAM NVRAM modules for journaling changes)

Hierarchy RAM-based Persistence: Overview

Background

Currently, the programming-language technology, Hierarchy, already has a feature called **Frictionless Persistence** that adds database persistence directly into a programming language. More specifically, we added a new, hierarchical data-structure called a **Matrix** directly into Java, and then allowed these matrices to act like databases. The overall idea is that by making a native, hierarchical data-structure into a database, programmers now have access to one of the easiest forms of persistence currently available today.

The way Frictionless Persistence works is when an application compiled using Hierarchy starts up, an internal, database-server is also started as well. Then, when the application accesses one of its persistent matrices, all (or a majority) of this matrix's data is loaded into RAM. *This means that the application will access the matrix's data from memory, not by accessing in from the database server!* In addition, when the application modifies data in the matrix, it does so on the copy that's in its own memory. And, at the same time, this modification is performed on the matrix in RAM, this change is logged to hard disk (or, even better, to some form of non-volatile RAM) which is then sent by a background process back to the database server. The database server uses this update information to also make the change to its copy of the matrix.

The benefits of this form of persistence are:

1. The main benefit is to developers. Persistence has become much easier to work with. Frictionless Persistence *greatly* decreases the amount of code necessary that a developer would need to write to work with a database. It's like the difference in working in C instead of assembly language, but Hierarchy does this for persistence. It allows programmers to work with persistence in situations they never would have before because it was normally too much work.

2. Because matrices are stored in RAM, these persistent data systems is an order of magnitude faster than systems that use traditional database-access technologies.

   And, it is still even faster than working with other, RAM-based database-technologies like in-memory database. The reason is with an in-memory database, the persistent data is still separate from the application data. This means when the system accesses data from the database, it needs to, first, retrieve the data from the database, and then, copy this into the application's memory space. This collecting and then copying of data is relatively slow.

   In Hierarchy, the application is working directly on the persistent data, working directly with the persistent matrices. The application can process the data *as it iterates over the matrices.* In other words, query and data processing are not two separate steps, they are combined in one. And, on top of this, no copy operation is necessary to move the data from the database to the application memory space.

It is important to note that what makes this form of persistence practical is the fact that most of the persistent data is loaded into RAM. This allows a developer to work on the persistent matrices with little regards to performance. If the entire matrix was *not* loaded in RAM, then the developer would constantly have to worry about whether each matrix access or matrix modification caused the system to wait as it accesses the backend database. It's this idea of loading the entire persistent-matrix in RAM that allows the system to work with persistent data in a practical way. It allows the system to work with the data at nearly the same performance as if it was working with non-persistent data, as if it was working with data that is only in main memory.

Frictionless Persistence already has been implemented and the previous information we just mentioned is background information for the rest document. Therefore, the focus of this document is not on this older information, but on the new features that will be added. The two main topics we'll be discussing are:

1. Allowing matrices to be *distributed* across multiple servers.
2. Adding Frictionless Persistence not only to matrices, but two other data structures and data types in the programming language. Specifically, *Frictionless Persistence will be added to the collection data-types and to the regular objects of a programming language* as well.

Next, we'll give an overview on how these two new features work.

Overview - Distributed Matrices

In Frictionless Persistence, an application server works with matrices that are replicas of what's on the database server. That means, on the application server itself, the entire database is being loaded into RAM. More over, this means the current client-server pattern for distributing database data (when only servers have the data, *not* clients) does not work in this situation. A different architectural pattern needs to be used. We'll describe this architecture here.

** Note: this section is primarily taken from the* **Design Patterns** *section of this document.*

- **Client-Oriented Distributed Persistence Architecture**
  - In the traditional architecture for client/server persistence, there is a central database server (or a cluster of replicated/sharded database servers) that a client application accesses. Generally speaking, if a client wants data from the database, it must send a query request to the server which performs the query and sends the results back.
  - In **client-oriented distributed persistence,** the clients have either a partial or a full copy on their own system. This copy can be kept in either RAM or on hard disk (or both). Now, when a client application needs information from the database, it can simply access its own local copy.
  - The benefits of this approach are:
    - Clients can achieve better performance, theoretically, over the traditional client/server architecture. The reason is because the client doesn't need to request its information over the network.
  - Possible problems of this approach are:
    - Clients require lots of RAM, which is more expensive than hard disks or SSD's. And, since the RAM capacity of systems is much smaller than hard disk, there may be problems fitting the entire DB in RAM.
    - Now, the clients must coordinate with one another to synchronize any updates that occurred to the data. For instance, if one client wants to write a particular data location, and must make sure its write does not conflict with any operations the other clients are doing to the data.
    - The scheme for keeping data consistent across clients (called a *coherency protocol*) can be very complicated for this type of architecture. It can also perform very poorly if a great deal of coordination and message passing is required between clients.

- **Client Locks for Shared-Data Synchronization**
    - In the **Client-Oriented Distributed Persistence Architecture,** if a client needs to place a lock on a piece of data so that it can perform synchronized writes without causing inconsistencies to occur, the client needs to send a lock request not just to the central database server, but to *all* the other clients.
    - The reason is because the data isn't centralized in the database server (as we've previously mentioned), but *each* client contains its own copy of the database. And since each client could possibly be reading or writing the same location, whenever a client needs to write to a data location, it needs to place some form of lock for doing so.
    - To see how this works, referred to the section, **Using a Combination of Memory-Status Directory with Memory-Status Broadcasting**. This section outlines the use case for how this occurs.

Overview - Frictionless Persistence of Collections and Objects

Frictionless Persistence will be extended, supporting not just matrices, but to the collections objects (arrays, lists, trees, hashes…) and the objects of the programming language itself. More specifically, for persistent collections, they'll work just like with persistent matrices: a developer can add an annotation to a collection which instantly turns it into a persistent database. Now, when the system starts up, this collection is loaded into RAM from the backend database-server, and the system does all its read and writes only to the RAM version. And, when writes occur, the Frictionless-Persistence background-process automatically handles sending any updates back to the database.

And just like with Frictionless Persistence in matrices, if a client is a part of a cluster of servers, when it needs to perform a write, it can send a write-lock request message to the other clients to gain exclusive access to a certain section of the collection. The client can now perform any reads and writes to the data location without the overhead of constantly having to notify the other clients of what updates it is making.

This concept of adding Frictionless Persistence to matrices and collections can be taken even further. We will also add Frictionless Persistence to the standard object. This is different than persistent collections, because a persistent object does not need to be apart of the collection or a matrix in order to be persistent. *Any* system of objects can become persistent.

Frictionless Persistence of objects has many applications, two of the key ones are:

1. **Object-Oriented Databases** – Having the objects of a programming language directly support persistence is really how object-oriented databases should have worked. It's what object-oriented databases strived to be. With Frictionless Persistence of objects, any system of objects can be turned into a database. The objects *are* the database.

2. **Fault Tolerance and Debugging** – If Frictionless Persistence of objects is enabled on a system and the system crashes, the entire state is already persisted, so this server can be restarted in the same state extremely quickly. And, because Frictionless Persistence is designed to have such little overhead, this form of fault tolerance will have little impact on system performance.

Read/Write Syncing Between Caches – Two Protocols

Hierarchy persistence will support many different types of coherency protocols, but there will be two extremes in types of protocols. They are **Client Locking Using Memory-Status Broadcasting**, and **Central Memory-Status Directory**. The one we will primarily use is a combination of these two. We will present all three protocols.

Client Locking Using Memory-Status Broadcasting

The client wants to write block or exclusivity on some data, broadcast intentions using a **client lock request** (or a **client exclusivity-lock request**) which it sends to all the other caches. It does this by placing a message on a **message bus** and having other clients "snoop" these messages (listen to the message bus for updates). The message bus is a high speed network that connects:

- all the processors on the server
- all the servers in the cluster

There are two types of locks on the data that a client can use to synchronize writing and reading with the other systems: full-exclusivity lock and the bumpable-exclusivity lock

*What is the difference between a **full-exclusivity** and **bumpable-exclusivity** lock?*

- For ***full-exclusivity***, the requesting server gets access to data *for long as it wants*.
- For ***bumpable-exclusivity***, the requesting server gets access to *as long as it can.* This weaker lock can be *bumped* another process once the data.

What are the potential problems of using memory status broadcasting?

- For broadcasting to work most effectively, it's best to have a message bus that is extremely fast, if possible as fast as the system bus connecting the CPU's cache with its main memory. Because every time a write operation is performed, a cache needs to broadcast this write to all the other caches and wait for a response. If the bus is too slow, the cache will be doing a lot of waiting. Also, for situations where a data location is read and written to by multiple processes, ping ponging can occur. In this situation, the wait times between operations can be quite severe.
- Another potential problem of broadcasting memory status updates is that the message bus can be overloaded with memory-status messages. And, the more cores, processors, and servers that are added to a cluster of connected servers, the more messages that are sent. This means the bus needs to be even wider, and the processor core or dedicated hardware used to process these memory status messages needs to do more work. Both of these need to be carefully chosen to correctly meet the performance requirements of the cluster.

Tracking the Status of Writes and Reads Using a Memory Status Directory

All writes and reads are broadcast to a central directory that stores this information. More over, all reads and writes need to check it this directory before every access.

The problem with using a memory status directory is that it requires all reads and writes to access directory. The bus connecting each cache to the directory must be extremely fast and able to handle a great deal of traffic. The memory status directory itself must be high performance, since it is the only directory for the entire cluster. And, it must be able to handle a large number of messages in parallel.

Using a Combination of Memory-Status Directory with Memory-Status Broadcasting

For Hierarchy persistence, our coherency protocol will use a combination of the previous two methods. Each cache (as well as the database itself) will have its own memory status directory. In addition, each cache will broadcast its write-lock requests to the other caches over the message bus.

- The cache that needs to do writes, will broadcast its write request over the message bus to the other caches.
- The other caches will receive the write-lock request, and record the request in its own memory-status directory. It will also invalidate the data location in the matrix, so any reads and writes to this location in the matrix will fail.

  So when the application tries to read this location, it will fail and this matrix access operation will automatically check the memory-status directory for the status of this data location. It will then see that the data location is being written to, and then either wait until the write lock is released, or it will send a request to the writing server to register for any updates to the location to be sent to this server.

- Again, ideally, the servers would have custom built hardware for the directory (to store the directory and/or process memory-status request), so that they can do data status lookups under one cycle.

Distributing Persistent Matrices and Objects: The Coherency Protocol

Overview

For a persistence technology to be truly useful, it needs support distributed computing. More specifically, it needs to allow a set of cached persistent-objects (this includes matrices and collections) to be replicated across multiple servers in a cluster. And, when a set of data is replicated across many different servers, one of the most important problems is how to keep this data consistent. For example, if one server makes a change to a specific piece of data, this change needs to be broadcasted to the other servers in the cluster. This allows the other servers to update their own cached versions of the data location.

This section describes the coherency protocol that will be used by default with distributed persistent-objects.

Goals for Protocol

1. **Optimize Reads –** Read operations should have little or no overhead when performed. This may be difficult in a distributed system because reads may need to check the status of a memory location before its read is performed.
2. **Optimize Private Writes** – How? With two types of write-lock. The first is a full write-lock, which gains exclusive access for an application instance as long as it wants. The second is by providing a **Bumpable-Exclusivity Lock**. Bumpable Exclusivity is a state that memory can be in that can be unlocked by other application instances. Other application instances can "bump" this lock at any time by simply sending a read or write request on the message bus. This form of write lock that takes advantage of temporal locality.
3. **Optimize All Forms of Writes –** In terms of priorities of optimizations, after optimizing for reads, optimize all forms of write operations. We want to make obtaining a write lock as fast as possible. What we need to reduce is the time it takes for the server to send out a write request over the message bus to the other servers, have it processed, and have a response return back to the original server. In addition, we also want to reduce the amount of bus traffic because as the cluster scales up with more and more servers and processors, this typically means a greater amount of memory-status messages that are needed to be placed on the message bus. It's important not to saturate the message bus as this will impact the performance of all the servers on the cluster.

   A two-part solution for these two problems is:

   a. Have a super-fast bus that connects the different servers.
   b. Have a known **worst-case latency for a trip for a message response**. This way, when the server sends out a message such as a write lock, the other servers don't need to send responses if this write lock has no conflicts. The originating server needs only wait for the time period of the worst-case latency and if it has received no messages, it knows the write-lock was a success.

Optimizing Reads

As previously mentioned, fast reading is the highest priority. We want to have the lowest amount of overhead added to a reading operation. So, when reading to a distributed persistent-object, only one check needs to be done before the read, *Is the Data Valid*.

The reason a read has to do an, *Is the Data Valid* check, on the data set is because this data location can be set to the invalid state if another application instance has obtained a write-lock on it:

Another application instance has made a write-lock request on this data set, and the memory controller process for this cache has marked the data set invalid. The memory controller has also added the write lock request to its memory-status directory.

There are two ways to implement invalid data:

1. Add an invalidation flag to the data structure itself
2. Add the invalidation flag for the data set to the memory-status directory – actually, the flag doesn't have to actually be added to the directory, having a write-lock entry created is the same thing.

Optimizing Private Writes

As previously mentioned, we want to optimize the situation where one application instance is allowed to read and write a data set without having to constantly check the status of the data set or to constantly have to negotiate for write-locks. So, what we need to support is *private access to data*, which is where an application instance request some form of exclusivity on the data set before working it. This exclusivity means when other application instances need to read or write this data set, they cannot do so without first negotiating with the holder of the data.

There are two forms of exclusivity box:

1. **Full-Exclusivity Lock**
2. **Bumpable-Exclusivity Lock**

We **will** describe both here.

Full-Exclusivity Lock

In a Full-Exclusivity Lock, the requesting application instance sends out a full exclusivity-lock request on the message bus. The receiving servers process this request by first invalidating the data set either by marking the data set invalid in its memory-status directory, or marking it invalid in the actual persistent object itself. Note that in the memory status directory of the server of the *requesting* application-instance, the memory-controller process should invalidate the data set on this server as well. The reason is because multiple application instances can be running on the server which can also try to read or write this data set.

Now, if another application instance needs to read or write this invalid data-set, when it performs this operation, it does an invalidation check and will find that this data set is invalid. Seeing that this data is invalid, the application instance checks its server memory-status directory on the data set, and will find that there is a full exclusivity-lock. At this point, it waits until the lock has been released. It can do this by registering itself in the memory-status directory as a watcher on this lock. And when the lock-release message is sent by the originating application-instance, this message is received by the memory controller which removes the lock from its memory status directory, and then notifies all the watchers of this lock that it has been released. This allows them to continue to perform whatever operation they had needed.

The benefits of this lock is that if an application instance needs exclusive access to a data set, it can do so using this lock.

Bumpable Exclusivity-Lock

The way the Bumpable Exclusivity-Lock works is it's very similar to the full exclusivity-lock, as it sends out a lock request to the other servers, which is then recorded in each server's memory-status directory. But the difference is if another application instance needs to read or write the data set and sees in its memory status directory that the data set has a bumpable exclusivity-lock, it can immediately send a bump request to the lock holder. This request orders the holding application instance to release the lock immediately (or as soon as it finishes its current operation) and for it to send a release message as soon as the data set is free.

The situation when a Bumpable Exclusivity-Lock is used is when an application instance is reading and writing some data, but this data is not involved in something that requires the stringent exclusivity of something like a transaction. This application instance doesn't need to keep others from accessing the data, but it still needs to read/write the data with a degree of synchronization.

Moreover, the reason this lock is so useful is because it's taking advantage of the fact that most data is worked on in a fairly isolated way (temporal locality). This lock allows an application process to work on a data set in a private way, without having to constantly refer to its memory-status directory or coordinate writes and reads with other application instances. But, the lock still allows other application instances to gain control if needed.

Other Optimizations

- **Server-Scope Multi Write-Lock**
  - If a server detects that two or more cores/processors on the same server are trading write and reads, causing the state of a data set to changed back and forth, then, the memory controller can upgrade this write lock in the memory-status directory to a **Server-Scope Multi Write-Lock**. This means that multiple cores/processors on the same server are sharing access to the same data.
  - This is more efficient, because now, the server doesn't need to broadcast each lock change to the other servers, since all the writes are happening on the same server.
  - In addition, the locking mechanism that's used can just be the standard, Java locking mechanisms normally used when concurrently accessing a shared variable.
  - In fact, a server can detect if multiple cores on the same processor are trading write locks. In this case, the write lock is upgraded to a **Processor-Scope Multi Write-Lock**. Here, the value is being shared in the processor's shared-cache (typically, the L2 cache), and its locking can be done in this cache as well.

- **Optimizing Access of Highly Shared Read/Write Data** (ping ponging)

  - **Shared Cluster Cache** (Optional) – The memory buses of all the servers are connected to one shared-cache. This shared cache is used for data that is read/written by multiple processors.
    - The servers in a cluster can automatically detect if a data set is being shared across multiple servers, and then move this data set to the shared cache.
    - A data set can also be manually moved into the shared cache by using the persistence API and/or some form of annotation marking this data set as shared.
    - The ideal performance of this shared cache is to operate as fast as the server's main memory. This way, exchanging read/write operations between application instances on different servers would happen as fast as they would if they were on the same server. This means that ideally, an extremely fast bus would be connecting the server to this shared cache memory.

  - **Ping-Pong Detection** – A process running on a dedicated processor (or dedicated hardware) on each server snoops the read/write operations in the cluster, and detects when a data location is being ping-ponged between two servers. If it detects this, it moves this data to the shared cluster cache.

    - **Branch-Prediction Programming** – Instead of waiting for the results of the locking request, the server simply sends the request and guesses at what the response will be. It will continue to process based on this guess. And when the response finally arrives, the server simply continues to process if its guess was right, but if this guess was wrong, it will roll back whatever changes it made and restart its processing.

- **Optimizing the Flood of Write-Requests Messages That Are Sent over the Cluster Bus As the Cluster Is Scaled up** (Fast Bus)
    - TO DO!!!

Design Patterns Used for RAM-based Persistence

- **Client-Oriented Distributed Persistence Architecture**
  - In the traditional architecture for client/server persistence, there is a central database server (or a cluster of replicated/sharded database servers) that client applications access. Generally speaking, if a client wants data from the database, it must send a query request to the server. The server processes this query, and sends the results back to the client.
  - In the **Client-Oriented Distributed Persistence Pattern,** the clients have either a partial or a full copy of the database on their own system. This copy can be kept in either RAM or on hard disk (or both). Now, when a client application needs information from the database, it can simply access its own local copy.

  - The benefits of this approach are:
    - Clients can achieve better performance, theoretically, over traditional client/server architectures. The reason is because the client doesn't need to request its information over the network.

  - Possible problems of this approach are:
    - Now, the clients must coordinate with one another to synchronize any updates that occurred to the data. For instance, if one client wants to write a particular data location, it must make sure its write does not conflict with any operations the other clients are doing.
    - The scheme for keeping data consistent across clients (a *coherency protocol*) can be very complicated for this type of architecture. It can also perform very poorly if a great deal of coordination and message passing is required between clients.

- **Client Locks for Shared-Data Synchronization**

  - In the **Client-Oriented Distributed Persistence Architecture,** if a client needs to place a lock on a piece of data (so that it can perform synchronized writes to this data set without causing inconsistencies to occur), the client needs to send a lock request *not* just to the central database server, but to *all the other clients*.
  - The reason is because the data isn't centralized in a database server (as we've previously mentioned) as each client contains its own copy of the database. And since multiple clients could possibly be reading or writing the same data, whenever a client needs to write to a data location, it needs to place some form of lock before doing so.
  - To see how this works, referred to the section, **Using a Combination of Memory-Status Directory with Memory-Status Broadcasting**. This section outlines the use case for how this occurs.

  - **Requirements for successful client lock protocols:**
    - A reliable message bus
    - Reliable clients

    Because if a client request is unable to be sent to a server in the cluster, or server is slow in responding, and duplicate locks by different clients could occur.

  - **Problem – How do you deal with overlapping lock requests to the same data?**
    - **Solution 1** – One system acts as an arbitrator, which detects the two overlapping writes and broadcast the correct ordering.
    - **Solution 2** – All the system clocks of each client are synchronized so that all requests have some form of timestamp or counter added to each request that indicates the ordering.

- **The Reservations Pattern for Transactions**
    - When creating transactions, instead of locking resources at the beginning and then holding them throughout the entire transaction, *reserve* the resource.
    - A system reserves the resources it needs at what would normally the beginning of the transaction. This means that a thread or application only has the resource locked for a short time. This allows other threads or applications access to the resource instead of holding on to them the entire length of this transaction.

    - **For example** – If a user is buying multiple products from a shopping website, the site should initiate the user's purchase by, first, reserving each item that the user needs, decrementing the product count for each one. Then, performing the rest of the transaction.
        - It only locks each item count variable for a short time, allowing other purchase transactions to access them immediately afterwards. Then, when the user finalizes his purchase, the reservations are fulfilled.

    - **Note: this is an existing design pattern**. It can be found online by doing a search for "reservation pattern."

Frictionless Persistence and Collections and Objects:
The End of the Database Era

Background

The idea of sending a query to a centralized database is beginning to come to an end. This older model is too cumbersome and requires devs to handle too many issues such as performance and also translating database data into the objects defined in the system. Devs want to work with persistent data as simply as possible. Currently, we can see this larger trend through a few, smaller trends:

- The movement towards in-memory databases
- The rising popularity of NoSQL databases – The structure of data in a NoSQL database as much closer conceptually to how developers work with data. More specifically, hierarchical data is much more natural to developers than relational data. Hierarchical data is naturally how we structure the objects in object-oriented system.
- The continued popularity of embedded query-languages like MS LINQ

All these smaller trends point towards the idea that developers want to work with persistent data much more naturally with much less thought for technical details and we currently do. But, the reason we stick with using remote, centralized databases is because:

- To work more naturally with persistent data, applications would need to be able to fit most of the database into RAM. But in the past, databases were too big to fit into a cost-effective amount of RAM.
- And, even if the databases could fit into RAM, hardware made these types of servers less practical because the cost of multi-processor/multi-core systems was prohibitive – Multi-core/multi-processor systems are necessary if the database is stored in RAM, because the server must be able to handle multiple parallel requests.
- It's tough to distribute in-memory data – for instance, synchronizing logged writes can be very complex.

But, nowadays, there are many consumer and business servers that can be purchased for relatively cheap (tens to hundreds of thousands of dollars) that can accommodate enormous amounts of RAM:

- Dell PowerEdge C8220 – Max RAM 256 GB
- IBM Power 795 – 8TB (that's *terra* bytes)

These newer servers can easily fit most databases in memory! This is why in-memory databases are becoming such a big deal these days. But, in-memory databases and even NoSQL are just stepping stones towards a larger trend. This larger trend is that developers want to work with data as naturally as possible with little concern for issues like performance and translating database data into objects. The goal is for devs to work with persistent data using objects that they are already used to and that for these objects to be pulled directly from RAM

*Note: developers *are* able to work with persistent data from RAM, with products like in-memory databases and memcache, but the developer has to *add* these solutions to his system. Working from RAM with persistent data should now be the default!

Solution

All data collections will support frictionless persistence.

More specifically, just like persistent matrices, persistent collections will work in a similar manner where a developer adds an annotation to a collection which instantly turns it into a persistent database. Now, when the system starts up, this collection is loaded into RAM from the backend database server. The system does all its read and writes only to the RAM version, and frictionless persistence automatically handles sending any updates back to the database server.
And just like with Frictionless Persistence in matrices, if a client is a part of a cluster of servers, when it needs to perform a write, it can send a write-lock request message to the other clients to gain exclusive access to a certain section of the collection. The client can now perform any reads and writes to the data location without the overhead of having to notify the other clients of what updates it is making.

What's interesting is that querying data can now be done in a way that's very natural to a programmer. Before, when a dev wanted to grab a set of data from a persistent data source, he/she would write a query in a separate language, having the system send this to a server. The system then waits for the results and only can beginning processes them when they've been received. Now, programmers can query data by using the standard techniques that they are used to: they can search through their collections using different types of loops (like `for` loops).

And even more than this, the concept of querying doesn't really fit this new way of working with persistent data. Really, the developer is simply working with native data structures that are persistent using *standard programming techniques*. They aren't really doing any type of query at all; they are simple working with these data structures as they normally would work with a regular (non-persistent) data structure:

```
for (int i = 0; i < myPersistentListOfContacts.size(); ++i) {

   //  Reading a persistent collection is done no differently than reading a
   //  regular collection
   String name = myPersistentListOfContacts.get(i);

   if (name.equals("Herbert Hoover")) {
      System.println(name);

      //  Writing a persistent collection is also done no differently than writing
      //  a regular collection
      myPersistentArrayList.add(i, "Hahn Solo");
      ++i;
   }
}
```

As you can see above, when a dev works with a persistent collection, he can mix read and write operations as needed. Now, instead of "querying a database," devs simply just access the collection. And, to do an "insert," they simply use the collection's native add/modify/delete methods and operators. Note, this also means that reading data ("querying") and writing data ("insert") are not separate activities like they are when a developer works with a database. Developers can easily mix the two as needed.

Persistent collections will support the following features:

- **Storage of Objects** – Persistent collections will be able to store the native objects of a programming language. These objects can be stored back to the back end persistence server. And, so that these stored objects can be accessed using other programming languages that have had Hierarchy Persistence added in, persistent objects will support the following features:
  - **A universal format for objects not specific to Java**
  - **Standard libs for common data objects** - for example, dates, network ports, strings…

- **Nested Persistent Objects and Collections** – Persistent collections will be able to nest other persistent collections. And, persistent objects will be able to nest other persistent objects. Persistent objects will also be able to nest persistent collections (and vice-versa). What this all means that any combination of nested persistent-objects and nested persistent-collections as possible.

A LINQ style query language – Again, a separate query language is not necessary to work with persistent collections, but maybe useful in many situations. This LINQ style query language will incorporate map-reduce features.

- **Multilevel Caching** – Just like with Frictionless Persistence of matrices, persistent collections will support multilevel caching and HSM.
- **Distribution of Shared Collections and Synchronization –** Like persistent matrices, persistent collections can be automatically distributed between clients, and can use locking and syncing mechanisms to coordinate reads and writes to these shared collections by the different clients.

Persistent Objects

Premise

Any system of objects can become persistent. This is different than persistent collections, because persistent objects do not need to be apart of the collection in order to be persistent. And, what we mean by a “system” of objects is that an object often has nested objects through the aggregation relationship (by including an object is a member or variable of a parent object).

Persistence Characteristics - The different types of persistent objects have different types of persistent characteristics based on how they are used and how they behave. Two main classes that we have seen so far are:

- Persistent Collections
- Data Objects

Each type is treated differently in terms of persistence. And, developers can create their own persistent data-objects and their own custom persistent collections. They can use a persistence API to customize their persistence behavior.

How is this Useful? In Two Ways

1. **Object-Oriented Database –** This persistence scheme for objects and collections is how object-oriented databases should have been done. This is the natural way to work with object-oriented persistent data. A possible reason why OO databases were never done in this manner was because for this type of object-oriented persistence to be practical, most of or the entire database needs to be stored in RAM. This allows persistence to be “seamless” with its objects.

2. **Fault tolerance** – When a system crashes, the entire state of all its objects has already been persisted to the database server. This is useful in the following ways:

   a. **Restarting the server in the same state –** While the system is running, all the objects can be constantly persisted back to the database server. And then, if the server crashes, the persisted state can be used to restart the server where it left off. Or, what might be more likely is, there may be servers in reserve that, once a server crash is detected, one of the reserve servers is pulled up and the state of the crashed server is immediately loaded into reserve server. This reserve server takes over where the crashed server left off.
   b. **Debugging** – And, when a server crashes, not only can you recover, but system administrators can use the persisted state to see what the server was doing when it crashed. It can even use the state to replay what happened.

Persisting Other Parts of a Programming Language

Note that anything that is normally used for persistence by the system can be modified to use this form of persistence scheme. For instance, memory-mapped files can be modified so they operate as fast as RAM (and not as fast as disk as they currently are set up to do!).

Four Main Types/Scopes of Persisted Objects

Description

Persistent objects can have different scopes, and by having a different scope, the persisted object typically has a different role that it plays within the system (Note, these scopes apply not just to objects, but to matrices and collections as well). The four different types/scopes of persistent objects are:

2. **Universally Shared Objects** – These persist objects are shared across all systems. These objects are most often the role of database objects.
    a. Universally shared objects can also be in the role of **Software Agents**. Software agents are components that act as middlemen between applications. Because of the usage of write locks, these agents could be fairly good at real-time transactions, but they would be even better as an event driven software agent. In situations where an application instance would register itself with an agent so that it can be pushed updates to some resource.

3. **Application-Cluster Objects** – A persisted object can have the scope of being consistent across an entire server cluster. This is useful in situations such as if you want to have a second-level cache that is shared across all the servers in a cluster. The data in this second-level cache would have the scope of the application cluster.

4. **Application-Instance Objects** – The scope of these objects are limited to a single, application instance. They are mainly used to save application state.

5. **Remote Application Instance Objects** – You can use these persistent objects so that you can do RPC-type calls to objects on another system.

Hierarchy Feature List

Data Properties

Definition

- Data Properties refer to the concept that subsets of a dataset/object can be assigned different data-properties. For example – subtrees (or, for objects, whole classes with their nested, aggregate members) can be assigned properties. Data properties describe how this piece of data will behave, typically in terms of persistence.
- Examples of data properties are: **Eventually Shared / Immediately Shared, Exclusive…**

- Individual properties can be assigned all the way down to each field in a matrix, or variable in a class.

Major Sets of Data Properties

- **Data Sharing policy** – describes what type of sharing this data set will do with amongst all the clients. The different types of this property are:

    - **Eventually Shared** – If data is eventually shared, any changes to the data (writes) will be allowed to trickle down through the system using *Client Broadcasts of Updates,* which means status changes to pieces of data are broadcast to all the other caches, clients, and the backend DB server. This update will be processed by all the other clients in their own time.

    - **Immediately Shared** – When data is immediately shared, then some form of locking will occur on the data set. This can occur through:

        - *Client broadcast of write-lock requests*

      Or if data is either: **Only Allowed to Reside On Server** (or another term for this is **Server is Bearer of Truth** for data), then:
        - *Lock requests managed at DB Server*

    - **Immediately Shared to Subset** – The immediacy of sharing occurs to just a certain subset of cores / servers is sharing data. For instance, a subtree of data could be only immediately shared to all the other servers in a cluster (and *eventually shared* to the servers outside the cluster).

- **Data "Bearer of Truth" Level** – determines where the "bearer of truth" for the up-to-date version of the data will reside.

  Levels:
    - **Application-Cache Level** – The client caches will have the most up to date version of the data. In terms of cache theory, data using this type of caching is using a *write-back* cache scheme.

    - **DB Server Level** – The DB server will have the most up to date version of the data. Two types of DB Server Level data:

        - **DB Server Level, client-cacheable** – Here, if a data set is **Immediately Consistent**, the clients can only have a *read-only* version of this data. If a client wants to write this data, it needs to send this write request back to the DB server first, which then issues write request to all the clients to invalidate the copy in all the clients.

        - Performance tuning – actually, to increase the performance of this operation, if the write request is coming from a client (not from the db server), the requesting client can send the write lock request to the other client caches at the same time it sends it down to the DB server.
        - For eventually consistent data, the client could also do writes as well as reads to its copy of the data

    - **DB Server-Level, non-client-cacheable** – Data that is only allowed to reside on the DB server. This means even for read requests, the clients must ask the server for the data. This would be closer to the traditional client/server DB model.

      This data-level is good for data that is going to be written using a server-side transaction.

Client-Pushed Server-Methods for Transactions

The way this works is a dev can write his data access method in the client application. This method must be static and not access any static/global variables. On compilation (or on system startup), the code for this method is pushed to the database server. So at runtime, when this method is called, the code is not run in the client, but called on the DB using some form of Remote Procedure Call.

Probably, most transactions will occur at the clients, using write locks. But, for data that is set to reside at the server-level, it's more efficient to do this change on the DB server. This way, the client is not adding these round trips across the network to perform locking

- Actually, as seen above, the client can send the request for the lock request to the clients at the same time it sends the request for the lock to the DB server, so maybe this isn't a big deal.
- But, Server Methods are still probably more efficient that client methods for data that is set to "non-cacheable" server-level.

Delayed Consistency using Branch-Predictive Programming: Another Mechanism for doing Transactions

The basic idea is for a section of code that often has to wait for some result (for example, a transaction where the client has to lock the data first before it can proceed with processing it), this code block can continue to be performed without having to wait for the return of the results. The system can guess what the value of the waited results will be and instead, continue to process. Then, when the system reaches a certain point in the code, it can wait for the actual result to return (actually, the results may have already made it back by that time) and if these results are the ones anticipated, the system can simple proceed on and continue to process. If not, the system can rollback the changes and redo the execution of the code block.

There are two different mechanisms that can be used for branch-predictive programming:

1. **Send waited-for-data at end of transaction**
   The client performs a transaction on the *copies* of the data in the cache and once the transaction is done, *then,* it sends the updated data to the database. The client then checks the values of the transaction, and based on these values, if it thinks the transaction will succeed, it continues to process based on the guessed results, performing the **predicted-branch code-block** on copies of the data (or simply by keeping track of the changes) – working on copies of the data allows the transaction and the predicted branch to be rolled back if the result is failure or different than expected.

At a certain point (for instance, if the client is ready to finalize the purchase), the client will block until the status of the transaction is received.

The different statuses of the results that are possible to be returned:

- If the status is ok, then proceed
- If failed, then rollback.
- Also, can receive 'ok with updates' – The transaction succeeded, but some of the expected values are stale. The DB server can send the client updated values, with which the client can update its processed data.

2. **Send waited-for data as it is encountered**
   In this type of predictive programming, *as soon as any write-locks or write updates are encountered, it sends these requests right away*. Again, instead of waiting for the results, if the client thinks it knows the correct results of the requests, it continues to process. And, if the results come back with success, it continues to process. If the results comeback as failure, then the transaction (and / or predicted branch code-block if the client has reached this stage) will be rolled back.

   NOTE: This is actually the type of Branch Predictive Programming that will probably be used more often.

Predicting the Branch Outcome

To predict the outcome of the branch, the system has a small evaluation routine written by the dev that, based on the current input values, can guess at the likelihood hood of success for the operation and the possible results.

For example

```
if (productCount > 3 && productPurchaseVelocity < LOW_PURCHASE_RATE) {
    predictSuccessfulTransaction = true;
} else {
    predictSuccessfulTransaction = false;
}
```

Interrupting processing on return of results

When the results return from the server, the client can be interrupted based on the outcome of the results. Especially for failure, the client should be stopped, so it can rollback the transaction and restart it.

What situations is Branch Predictive Programming good for?

Good for ecommerce, where most of the transactions will be successful (maybe 1 in 1000 have a problem – and these problem ones can be detected early and processed without branch prediction).

What are the benefits?

The client doesn't have to wait for the backend server to respond to resource requests to continue processing. This means:

- The server is more responsive
- Has better CPU utilization – requires less servers

Architectures

- **Clients with a central DB server** – The first type of architecture is where all the clients must write back their changes to the cached version of the data back to a central DB server
- **Clients with out a central DB server** – Here , there is no central DB server, the clients not only have the DB in RAM, but are writing the data back to their hard disks (or, maybe just a subset of servers in the cluster are). And, if a server crashes but needs to get back up to date on restart, it can query the other servers for updates.
- **Optional: Multi-level Caches** – both the above schemes can have multiple levels of caches below the clients. These caches can replicate data that's being held by the clients, or have the only copies of this data and the clients must refer back to the cache to get the data.

Statistics Gathering

Caches will gather stats on cache misses and hits, on write-lock problems (ping ponging). It will use these stats to optimize cache behavior.

Each cache can monitor itself and generate its own stats. Every once in awhile, a background thread will fire up and analyze the stats. Using rules and patterns detection techniques, this thread will identify performance problems and create solutions. Many of these solutions will be performed to the system in an automated fashion while the system is still running.

Fast Message-Bus & Dedicated Memory-Status Controller

Between servers in a cluster, there can be an optional, ultra-fast message-bus that servers can use to send messages on, such as write-lock requests, write-lock denials, and write-lock ordering… The goal of this bus is to operate at the same speed as that of a memory access by a CPU (or even as fast as the CPU's access of its cache). This way, a CPU will not have to wait a large number of cycles to do a write using a write lock. Ideally, a write lock and write should take two memory access cycles, maybe even one!

This means that the latency to broadcast a message to all the servers should be known (worst-case message roundtrip latency – which includes the time to process the message). This also means the time it takes to process a message by another server should also be known.

Message processing can be handled by a dedicated core/ CPU or even dedicated hardware custom designed to process these responses. This message processing unit does the following activities:

- Listen on the bus for memory requests and updates
- Checks and modifies the Memory-Status Directory based on the messages
- Modifies the matrix in RAM in response to the messages
- Generates and sends responses

Super-Fast Persistent-Memory Cache

Each computer should have a bank of super-fact persistent-memory that operates as fast (or even faster) than main memory. Its purpose is fault-tolerance, so that if a CPU performs what should be a persistent write to main memory, this write is also journaled in the super-fast persistent memory. This way, if the server dies, the system can restart still having the latest writes it performed written back to the main DB server.

Can use NVRAM technologies (non-volatile RAM) like FeRAM (which is very expensive but performs at near DRAM speeds), but actually, what would be more cost effective is to use hybrid persistent DRAM. This form of memory has DRAM, but in the event of a system crash, it has a battery which allows it to continue to operate and write its contents to flash memory.

Cache Modes

Matrices will have two modes they can operating:

- **Full Database Mode –** will cache the full matrix in RAM.
- **Cache Mode** – will cache only a part of the database in RAM (typically, the most frequently used data).

Cache Mode

Cache mode is complex, because querying is tricky. One of the main problems is if you run a query on a matrix that is only partially loaded in memory, how does the system know if your query is covering everything it should? This means really, cache mode is not as useful, because if you want to be able to search on just about any part of your matrix, you need almost the whole matrix in memory.

Full Database Mode

Here, each matrix in RAM is nearly a full replica of what's in the database. Note that this may not be as useful for Big Data applications.

Cache Hierarchy and Hierarchical Storage Management

Cache Hierarchy

Hierarchy persistence will support cache hierarchies, each cache level with a different speed. For instance, for a site like Amazon.com, the first level cache would be the matrices in RAM. The second level cache would be the most commonly accessed client-system product info. This second level cache would probably exist on its own server. The bus between the servers and the second level cache could optionally be a super fast network, just like what is described above for the connections between servers.

More specifically, caching level is based on latency (access time) and transfer rate

- **First Level Cache** – is on the client systems. They are *server level caches*.
- **Second-Level Cache** – is typically a separate server connected over a high-performance network. This separate server is a part of the cluster, and is a *cluster level cache*.
- **Third Level Cache** – is typically a global cache in front of the database.

Also, can create **synchronized data-blocks** at any level. These are groups of data (typically a subtree) that needs synchronized updating. For example, *product quantity* could be synchronized across multiple servers.

Hierarchical Storage Management

The different caching levels will support hierarchical storage management. This means that based on certain situations, a block of data can be evicted from the cache and moved to a lower level. The opposite is true; a group of data can also be automatically loaded into a higher level cache based on certain situations. This group of data can be defined as a specific set of items, or define by query.

And, hierarchical storage management can be extended past the database server, supporting common hierarchical storage management techniques where there can be slower, less expensive forms of storage underneath the database. For instance, there could be a main database that uses more expensive flash disks, and then beneath this server is another database that uses hard disks. This lower-level database can be an archive of older items. And based on the age of an item of information, items from the main database can be evicted and moved into the archive.

Query Pruning

Query pruning occurs whenever a query is run on a matrix that is cache mode. Query pruning determines how much of a new query needs to be pulled from a lower-level cache or from the database. This is a very tough problem, especially with hierarchical database, because hierarchical databases typically need good portions of the database to be in memory. This means that cache mode is not as useful. This also means that, typically, matrices will run in full replica mode, which additionally means that query pruning will not be necessary. But in certain situations, cache mode and query pruning will be useful.

Different techniques in query pruning:

- Query matching – the current query matches the previous query, don’t grab new data, use what’s in the cache.
- Query matching with subset detection – use rules and pattern matching to detect if the current query is a subset of a previous query.
- Run query on cache and then deter me what other data is needed.

If a query does not match any of the queries in the query-match cache, the query can be sent to the database, which also stores the set of previous queries run on the matrix server. The database runs the query doing its own pruning and only returning what is new back to the matrix.