CORNERSTONE
MINNESOTA STATE UNIVERSITY MANKATO

Journal of Undergraduate Research at Minnesota State University, Mankato

Volume 6 Article 21

2006

Implementation of a Segmented, Transactional Database Caching System

Benjamin J. Sandmann
Minnesota State University, Mankato

IMPLEMENTATION OF A SEGMENTED, TRANSACTIONAL DATABASE CACHING SYSTEM

Benjamin J. Sandmann (Computer Science)
Ann Quade, Faculty Mentor (Computer Science)

Research on algorithms and concepts regarding memory-based data caching can help solve the performance bottleneck in current Database Management Systems. Problems such as data concurrency, persistent storage, and transaction management have limited most memory cache's capabilities. It has also been tough to develop a proper user-oriented and business friendly way of implementing such a system. The research of this project focused on code implementation, abstract methodologies and how to best prepare such an application for common business usage.

IMPLEMENTATION OF A SEGMENTED, TRANSACTIONAL DATABASE CACHING SYSTEM

Abstract

Benjamin J. Sandmann (Department of Computer and Information Sciences, Minnesota State University, Mankato)
Dr. Ann Quade, Faculty mentor (Department of Computer and Information Sciences, Minnesota State University, Mankato)

Research on algorithms and concepts regarding memory-based data caching can help resolve the performance bottleneck in current database management systems. Problems such as data concurrency, persistent storage, and transaction management have limited the capabilities of most memory caches. It has also been difficult to develop a proper user-oriented and business-friendly way of implementing such a system. This project focused on code implementation, abstract methodologies and how to design such an application for common business usage.

Introduction

A database, or more accurately, a Database Management System (DBMS) has developed from being a useful tool, to a crucial component of any organization's success [11]. Companies are now able to store and manage gigabytes, and in some cases terabytes, of data in a well developed DBMS [11]. Because of this, the field of business intelligence, which covers data warehousing and data management, is becoming a major player in the technology world [3]. While the current DBMS architecture is extremely efficient and very capable of holding its own even in a high transaction environment, the gradual increase in dependency upon such technology is starting to show some weaknesses in current systems [13].

The goal of this project was to apply the concepts of caching to produce a practical solution to several of the issues mentioned above. Additional research was needed on applying the concepts of caching to a multi-tier and/or distributed solution. After analysis and review of the proposed and existing designs available, a new cache design could be formulated. This design would be more practical and feasible for common use. There would not be an excessive amount of assumptions made and the project would be configurable to the applications needs. Before discussing the methodology used in the design a little background information is needed [13].

Background Information

A DBMS should allow a user to efficiently manage large amounts of data, while upholding integrity and providing persistent storage [13]. The two most important

keywords in that concept are integrity and persistent storage. To uphold integrity, a DBMS has to have a multitude of concurrency methods [7]. Different locking strategies are used to ensure that data is always accurate and correct [7]. To allow for persistent storage, a DBMS keeps a master record or storage of all data on its system's hard drive. This means that every time a user wishes to query the data, the process will typically involve data being fetched from the disk. The issues described above are creating bottlenecks in most data solutions used today [11].

As I discussed above, a DBMS must keep some type of record of its data on a hard drive or some type of persistent device. When a query is run against a database, its instructions must first be processed and then the requested data needs to be fetched. It commonly takes anywhere from six to ten milliseconds to seek out the required data on a hard drive [9]. For most applications this is fine and any requested data, depending on the complexity of the query, should be returned to the user quickly. This is a relative statement, however, and will not hold true in large systems. If there are thousands of users querying such a system, the time required to seek and retrieve individual sectors of data is too large for an acceptable response time. Not only is the data retrieval too slow, but the time that locks must be held on records relates directly to the time of the transaction. If ninety percent of the transaction consists of writing or reading from the disk, the memory medium ends up being the main source of the bottleneck [13][11].

In reality, most bottlenecks are formed due to an overflow of network traffic. It should be easy to understand that if a DBMS is receiving more requests than it can process, the response time will gradually increase. Imagine if thousands of hungry people suddenly showed up at your favorite fast food restaurant. With only one line and two employees, how could they ever manage such a workload? Such situations create an environment in which most data solutions cannot keep up. Most companies have leased large mainframes to attempt to increase the processing speed of their system. Such systems are usually so expensive, however, that it is unfeasible for a company to have multiple mainframes. In reality, such systems only buy companies a little time until they gradually increase their workload to a point at which the mainframe can not keep up [13][15].

Research and Literary Review

So how does one go about providing solutions for these potential bottlenecks? It seems that research and industrial practices are leaning to a more distributed management system [12]. Instead of spending large amounts of money on massive mainframes to increase their system's processing power, companies are choosing to distribute the workload over several systems [12][4]. Clusters of workstations typically provide the same amount of throughput while typically costing much less than large mainframes. One reason for this is that the large cost of a mainframe isn't only due to a massive increase in processing power. The extra cost of a mainframe typically comes from its reliability and high availability [8]. However, there are software packages on the market today which can provide a sufficient level of high availability when using a distributed

approach [8][10]. This means that a distributed cluster can have many of the same benefits of a mainframe other than simply a boost to the overall processing power.

The battle between mainframes and distributed systems ties into caching as well. The general concept is to distribute the workload so that a greater overall throughput can be achieved. As with any cache, its fundamental speedup will be provided by the principal of locality [1]. This simply means that a system should do as much processing and data storage as close to the application as possible without compromising the integrity and availability of the system. The combination of clustered workstations and in-memory caches creates a very robust IT infrastructure. Developing a data caching solution in a multi-tier system, however, is not easily done. The concept becomes even more difficult when applied to a distributed approach, such as server clustering [1][6][8].

Design Methodology and Concepts

The first step in my design was to look where my requirements differed from other implementations. Unlike many of the cache designs that I was familiar with, I wanted to be able to retrieve records on a relationship or tuple basis. I also wanted to be able to retrieve these records with discrimination of certain attributes and their values. The problem with my project is that developing a SQL, or Structured Query Language, parser and engine was completely out of this project's scope. Because this problem was apparent from the beginning I was able to tailor my implementation to potentially fill this void. Before discussing the tuple based retrieval and attribute-value discrimination, the infrastructure of my design needs to be explained in further detail. Most of these concepts can be considered the 'core' of the system.

When I began the design and implementation of my core, I already knew three things.

1. My engine would be transaction based
2. I would need a loader to allow for communication with the back-end
3. I would need a basic system of garbage collection or data eviction

I will break this section up into three phases, and then cover the main infrastructure on the last phase, as it represents my final design.

Phase 1

The first phase of my system consisted of developing a trivial record-set cache. Basically an application would get a session from an instantiated cache and begin executing SQL statements. The cache would then use the loader, shown in Appendix B, to fetch the requested records from the back-end and simply store the record set in memory. The data would actually be hashed as an object via the original SQL string. When an application requests the same query I would hash the SQL string, and if a record set existed, it would be returned to the user. This would allow for a decrease in network traffic and would distribute more of the workload to the client computer. The client's overhead was actually relatively minimal, however, as the processing algorithm was extremely straight

forward. If the hashing returned a null result, it was evident that it had not recently retrieved that record set. I also wanted to note that even though the users were required to get a session via the programming model, it didn't actually do anything at this point. It was added due to the realization that I would later need to allow multiple users to operate on the same cache in a transaction-based environment.

There are a lot of problems and false assumptions with the design mentioned above. The first is the apparent assumption that the back-end data will never change. Such an assumption greatly compromises the consistency of the application's data. Although it is a very evident problem, let's look at a brief example in a step-by-step process. It is assumed that there is a central database that multiple applications use and there is at least one user using my cache framework.

1. The cache user initializes the cache and gets a session.
2. They begin the session, and execute a "Select * from TableX" statement.
3. Upon commit, the record set is fetched and stored in the cache.
4. Another user commits an update statement to the database on a record in TableX.
5. When the original application re-executes a "Select * from TableX" statement, the cache will return the hashed records, and the application will now be accessing invalid and stale data.

The problem is that there isn't any mechanism for detecting or getting notified of changes to the persistent data. Imagine if this project were applied to an Enterprise Resource Planning(ERP) system or a trading system. The very integrity of the application would be completely compromised. Possible solutions to this problem are covered in the third phase.

Another obvious problem can be seen from the example above. What would happen if a user were to execute queries that would return identical record sets, but whose SQL strings were different? Since the sets are hashed via the string, any differences will cause the hashing method to look in a different location [5]. Take a look at the following queries assuming that we have a table called TableX, which consists of attributes *id*, *fname*, and *lname*.

- Select * from TableX
- Select id, fname, lname from TableX
- Select * from TableX order by id asc
- Select * from TableX order by id desc

The first two queries in the list will return completely identical record sets. However, the cache will end up storing the set twice because the SQL string will have two different hash values. If you look at the latter two queries, they are identical in nature as well, but simply have ordering manipulations applied to them. Such a query would require unneeded network traffic. The ordering manipulations should really be done on the client side. Also, if you compound this problem with the inconsistencies mentioned above, you

now have four very similar record sets all being maintained in the cache. Each of them will most likely have a different version of the back-end data. The problems associated with this phase led to the changes made in Phase 2 of the design process.

Phase 2

This phase of the development saw the introduction of two new models, the evictor, and the loader. These models were added to alleviate some of the problems mentioned above as well as to add new functionality. In Phase 1, the loader/adapter, which communicated with the back-end, acted as a proxy. It would forward the request and cache the results. In Phase 2, the loader became completely configurable and provided the mechanism for invalidating stale data. This was required due to the multitude of design changes to the core.

In this phase, the core did not simply act as a record set cache. Instead it stored individual records returned by the loader. Each object would be hashed by its primary value, or basically a Java object representing its primary key. This allowed users to extract single records instead of having to parse or iterate over an entire record set. The way that these objects were cached was determined by the loader interface. The retrieval and insertion method for these records also changed. In Phase 1, the cache was populated by forwarding the SQL string, but now that record sets were fragmented, I decided to convert the cache to a map-based architecture. Records would be manipulated via *put*, *get*, *update* and *remove* statements. I have listed a brief end-to-end example below.

1. The user requests a session from the instantiated cache.
2. The user begins the session and manipulates the cache via get, put, update, and remove statements.
3. The user commits the session.
4. The cache buffer is applied to the back-end via the loader.
5. The loader successfully applies or rolls back any changes.

From the sample session above it is clear that the general format and design went through a huge overhaul in this phase. There was now a cache buffer in which all pending changes were applied. When a user manipulated the cache via *puts* and *updates*, the changes would get applied to only the buffer. The buffer then served as the first data lookup with any misses getting redirected to the cache.

I did not mention any data retrieval methods that related to the buffer for an important reason. Any get command or data retrieval would need to be immediate. The data retrieval would first look in the immediate cache buffer, and if a reference wasn't found, it would then look at the cache. The loader would then make sure the particular cache entries were not stale and would return the correct data to the user.

It was now that I realized that the loader would need to be extremely configurable. Since I was now populating the cache via the loader, and implicitly calling its methods upon data retrieval, I would need to allow the user to tailor the actual implementation to their

specific application. The user or systems administrator could then configure how they wanted the objects to be stored, arranged, and retrieved from the cache. The dirty work of handling the transaction, retrieving the data, and monitoring any concurrency controls or inconsistencies would be managed by the cache framework.

The last concept to be covered in this phase was the evictor. Basically when entries would get added to the cache they would be distributed over a set of configurable buckets. An evictor thread would be started and would monitor these buckets. When the eviction interval was up, the evictor would invalidate a set amount of data from the cache. The speed of the invalidation was dependant upon how many buckets existed and how long the eviction interval was set to. It was decided that all caches would use an LRU, or Least Recently Used, eviction algorithm. When an object was touched from an application, it was removed from the front of the queue and placed at the back. This ensured that only the most infrequently used data were getting evicted first. A diagram showing the functional and structural layout of the evictor can be seen in Appendix A.

Phase 3

This phase consisted of refining several concepts of Phase 2's design. While the core design definitely changed a lot, the focus was on adding features and functionality to make the product more useable. The changes to several concepts in the core are depicted in Appendix C.

The main changes to the core consisted of segmenting the cache. With the previous design, there was just a large cache of hashed objects. The user configured the loader to load specific objects representing database records into the cache. It was up to the user to provide a virtual mapping of this record to the table from which it came. In this design, however, there existed a CacheTable object which provided a direct relationship to that of the database tables. At first this design change was considered simply to provide a more organized mapping from the cache to the back-end, but later it was realized that this was the necessary route for other reasons as well.

This format makes it possible to properly segment and map the cache. The cache tables could then have their own loaders that would be responsible for loading its data only. This made the cache even more flexible as users would now have even more configuration options. When an application now interacted with the cache, it could directly manipulate these tables, and it would be possible for the administrator to know exactly how the changes would be replicated to the back-end. To better explain this I will run through a brief scenario.

1. A cache is requested from the global cache manager.
2. The user then creates specific tables which will then be used to map the cache's contents to the back-end.
3. The user will be passed back a TableConfiguration object in which they can configure plugins such as the evictor and the loader.
4. After creating the cache structure the user requests a session.

5. The application then requests proxy handles to the cache tables and begins to manipulate the table via *put*, *get*, *update* and remove methods.
6. Upon the committing of a session the transaction buffer will be applied to both the cache and back-end (if writes were involved) via each table's loaders.

While there are actually many more potential routes of execution, the example displayed above is the typical usecase scenario for an application. It should be noted again that the objects in which the user interacts with are actually proxies of the cache tables. This will be explained in more detail below.

Besides providing greater logic and a better, cleaner organization of the cache, the segmentation also allowed for better concurrent throughput. In Phase 2, a proper lock manager would be required to provide and monitor locks for all records in the cache. Not only was this out of my project's scope, but it would introduce a lot of potential overhead into the application of the transaction. Instead, I used a more trivial lock manager that allowed only one user at a time to obtain an exclusive lock on the cache. Any clients concurrently writing to the cache would have their sessions rolled back. In this design, however, I was able to develop a table-based lock manager. Now only one user at a time was able to obtain an exclusive lock on the CacheTable's proxy. This meant that if applications weren't using the same tables, there wouldn't be any issues with the simultaneous processing of their transactions. One thing that needs to be clarified is that the analysis which allowed for the granting of exclusive locks, actually monitored the whole transaction and not individual write operations. This meant that all of a pending transaction's lock requests needed to be fulfilled before it could be processed.

Another important modification to discuss was the segmentation of the transaction buffer. In the past, data manipulations were applied to a global cache buffer. Now they would be applied to a proxy of the CacheTable. This meant that the transaction buffer was now a set of table buffers. This allowed for a better organization of the pending transaction. It also greatly simplified the replicating of data to the back-end on a per-table basis. The segmentation of the transaction buffer also resulted in the restructuring of the rollback mechanism. In previous phases, if a rollback was requested, the session would analyze the pending transaction buffer and decide what steps needed to be taken. Now each table buffer contained a stack of the inverse data manipulations. This greatly sped up the process of rolling back transactions and also made its organization more structured. It should also be noted that the evictor plugins now operated on a per-table basis as well. The concepts and functionality were the same, but now only the individual table's records would be distributed over the evictor's buckets. The only real function-related change was in how the evictor removed the data. It now used a session and table proxy, so that its removal of the data didn't conflict with any pending transactions.

The last and by far the greatest feature to come out of Phase 3 was the introduction of indexes and index configurations. Since there was a more direct mapping to the back-end database, it was possible to have indexes for each table. A user could now request an IndexConfiguration object from the table proxy. This would allow them to configure

what objects were going to be stored in the cache table, and what attributes these objects would have. It would then be assumed that all of the objects would have getter and setter methods for each of the attributes listed. After records were loaded into the cache, each table would dynamically create an index on each of its record's attributes. This allowed for a host of new functionality. This concept was not only different from any design I had researched, but it provided the most functionality and greatest potential speedup to any application. After these indexes were built, an application could now query the cache with discrimination of the object's attribute-value pairing. In a sense, the object's attributes and values were now viewed as a single tuple. Using a getAll or removeAll method, an application could manipulate all objects with attributes equal to a certain value. Essentially this is the same functionality that a relational database can provide. This new component allowed the core code to avoid processing entire tables in the cache. The code could now reference the indexes and return the objects associated with the requested pairing or tuple.

All of the design changes in this phase were responsible for the framework now being a practical and useable product. It solidified the design and led to the eventual finalization of the implementation. The discovery of several bugs led to the addition of a few small components, but they were only minor parts of the overall design.

Benchmark Methodology

The two methods used for benchmarking my framework were relatively straightforward. A copy of "DB2: Universal Database" was obtained and used to represent the back-end database. The configuration settings were set to a typical system's settings and an 'Employee' table was created in the default schema. The table was then populated with 50,000 entries. This would serve as the data source for any queries.

The first type of benchmark was strictly designed to be a throughput and performance metricI wanted to see how many continuous queries could be processed by the system in a given time interval. To test this, a small set of queries were formed and executed in a random fashion against the database and cache. A sixty-second interval was set as the observation period. During the sixty-second observation period, the number of completed transactions would be recorded. After the data was gathered, the transaction versus time ratio could be calculated. It should be noted, that the first three minutes of the metric were unobserved to allow for Java's Just-In-Time compiler (JIT) to no longer affect the results of the test [4].

The second benchmark was designed to show the speedup provided by successive trips to the cache. As a cache becomes populated, its overall transaction times should lower. To test this, a small set of similar queries were formed and executed in a random fashion. The start time and completion times were recorded for each transaction, and the difference was then calculated. The queries would only be executed five times in succession. The test was then executed twenty times and the averages were calculated. The results of the benchmarks are displayed in Figure 1 and Figure 2 below.

Benchmark 1 Results

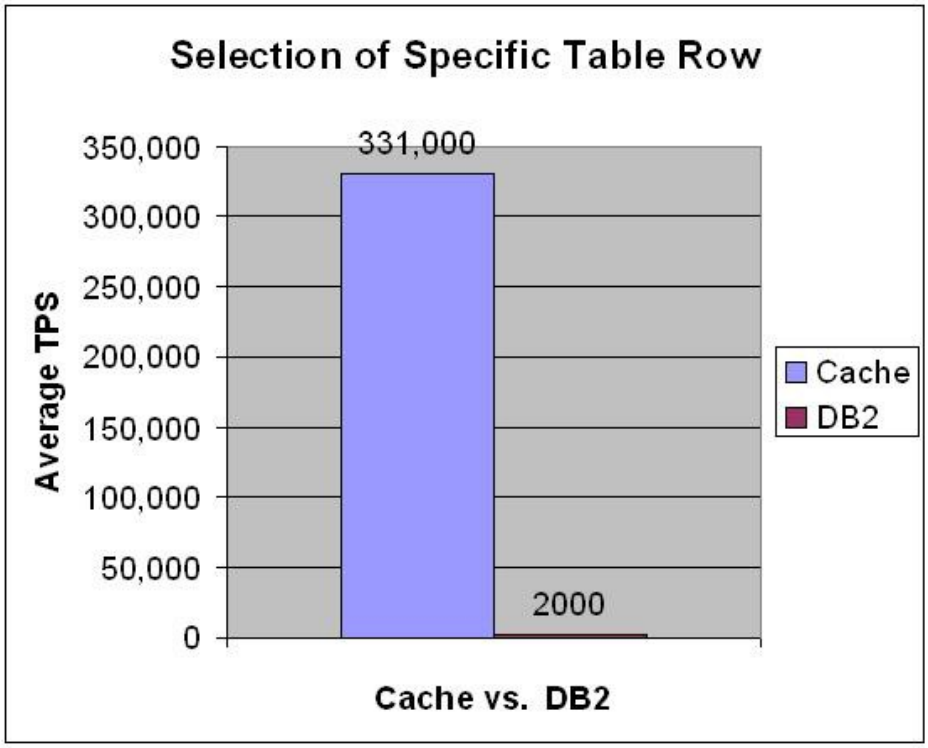


Figure 1: Average transactions per second for a cache-database pairing versus a stand-alone database

Figure 1 illustrates the difference in the average transaction per second between the two systems. The standalone database was able to process 2,000 queries per second whereas a database with my front-end cache framework could process nearly 331,000 similar queries per second. Although this type of result was expected, it came for another reason than usual. For example, the largest amount of speedup typically comes due to the principle of locality, but usually on the network level. However, in my testing, both applications resided on the same computer and therefore negated the potential speedup due to the decrease in network traffic [2].

In this case the speedup displayed in Figure 1 came directly from the difference in memory mediums. A typical hard drive will have seek times of 6-10 milliseconds [14]. Memory, on the other hand has an access time of around 70 nanoseconds [9]. A comparison of these times is shown in the example below.

Hard Drive seek time : 8 milliseconds, or $8 * 10^6$ nanoseconds
Memory access time : 0.00007 milliseconds, or $7 * 10^1$ nanoseconds

8,000,000 / 70 = 114,286

Access time to seek time ratio: 114,286 / 1

In other words, it is theoretically possible to make 114,286 trips to memory in the time it takes to locate one sector on a hard drive. Because of this and the hashing approach to the data retrieval, the in-memory cache provides an obvious speedup. Another important factor is the lack of any query processing. All of the cache manipulations are applied via the transaction buffer at commit time. There is no extra overhead due to query parsing, and the buffers that get applied are translated directly to hashing operations[13].

Benchmark 2 Results

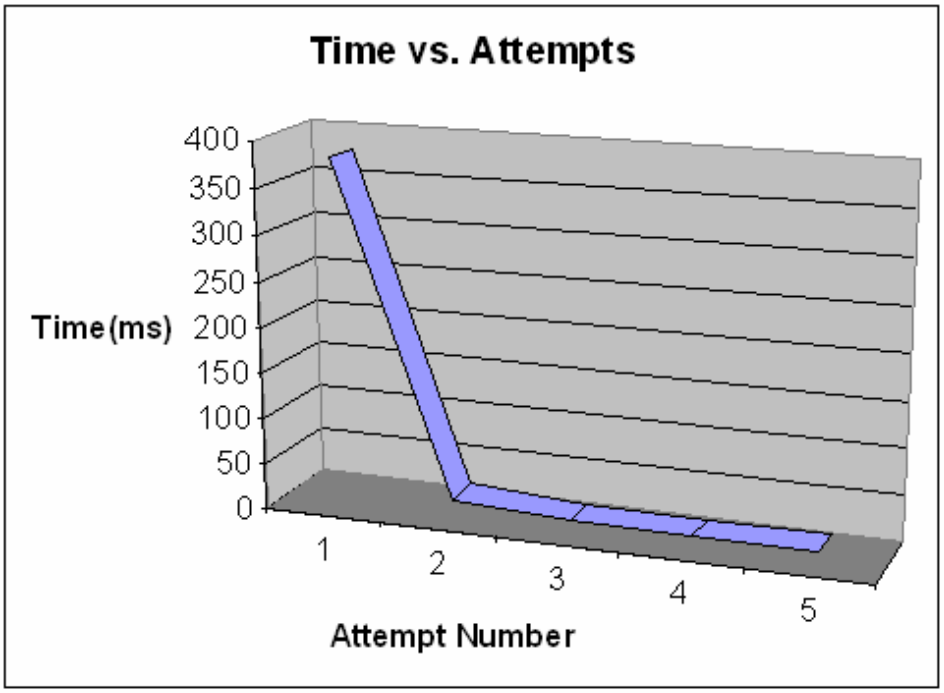


Figure 2: Average time required to process queries versus attempt number. The vertical scale is represented in milliseconds.

The second test was also successful and showed a relatively obvious speedup. Figure 2 depicted a huge decline in the query processing time from attempt one to attempt two. The reason for the slow processing time in the first attempt is due to the fact that the cache is completely empty. Any query will be forced to load data from the back-end. This will result in times similar to, and actually longer than, those of a standalone database. After the cache is populated from the first attempt, the access times drop dramatically. According to Figure 2, the time drops from roughly 380 milliseconds in the first attempt to only 10 milliseconds in the second. This is almost a 97% reduction in processing times. If a stand-alone database were used on the second attempt, there would

be some speedup due to partial caching by the operating system and database software, but it would not be close to the observed reduction in Figure 2.

Conclusion

From the results of the benchmark testing it is clear that providing a mechanism for caching data provides a potentially huge speedup over using a standalone DBMS. While there are many different methods of implementing a cache, the approach used in this project provides a practical and straight forward solution. It also uses several concepts and configuration options that are yet to be seen in other design proposals. Whether it is applied to a local application or distributed over a cluster of workstations, an in-memory cache can provide huge benefits. The difference in memory mediums provide quicker access times and less time required for locking, and the principal of locality and client-side caching keeps network traffic to a minimum. The DBMS is then free to focus on persistence and integrity instead of throughput. Due to its scalability, potential speedup, and plethora of benefits to the overall network stability, it is easy to see that distributed caches are the data management systems of the future.

References

[1] Keller, M., Basu, J. (1996). A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5, 35-47.

[2] Kurose, J., Ross, K. (2004). A Top-Down Approach Featuring the Internet. Addison Wesley, 3rd Edition.

[3] Loshin, D. (2003). Business Intelligence: The Savvy Manager's Guide. Morgan Kaufman.

[4] Wigglesworth, J., McMillan, P. (2004). Java Programming: Advanced Topics. Thomson Publishing

[5] Hamer, J. (2002). Hashing Revisited. *Proceedings of the 7th annual Conference on Innovation and Technology in Computer Science Education,* 80-83. ACM Press.

[6] Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B., et al. *Middle-Tier Database Caching for e-Business*. IBM Almaden Research Center.

[7] Thomasian, A. (1996). *Database Concurrency Control: Methods, Performance, and Analysis*. Springer, 1st Edition.

[8] DiMarzio, P., Watson, B., Ryan, P. (2005). *Scaling for High Availabilty: WebSphere XD and Websphere Appliction Server for z/OS*. IBM Corporation.

[9] Kozierok, C. (2001). Memory Acccess and Access Time. *The PC Guide*. Retreived May 18, 2006 from http://www.pcguide.com/ref/ram/timingAccess-c.html

[10] Roehm, B., Erker, T., Finneran, C., Klingensmith, K., Mann, V., Wan, K., et al. (2006). *Using WebSphere Extended Deployment V6.0 to build an On Demand product environment*. IBM Corporation.

[11] Hoffer, J., Prescott, M., McFadden, F. (2006). *Modern Database Management*. Prentice Hall, 8th Edition.

[12] Umar, A. (2003). *E-Business and Distributed Systems Handbook: Platforms Module*. NGE Solutions.

[13] Kroenke, D. (2004). *Database Concepts*. Prentice Hall, 2nd Edition.

[14] Kozierok, C. Seek Time. *The PC Guide*. Retreived May 18, 2006 from http://www.storagereview.com/map/lm.cgi/seek

[15] Yeargin, R. (2002). The Warmblooded Dinosaur : Linux on the Mainframe. Retreived May, 19, 2006, from http://librenix.com/?inode=51

Personal Biography

Ben Sandmann is a senior majoring in computer and information sciences with a minor in biology. After graduating from Cedar Mountain High School in Morgan, MN, he furthered his studies by enrolling at Minnesota State University, Mankato in the fall of 2002. Since then he has received several academic-based scholarships including the Federated Insurance Scholarship, the Department of Computer and Information Sciences scholarship, and the College of Science, Engineering and Technology scholarship. He was also a Who's Who Among Students in American Universities nominee and was awarded a Minnesota State University, Mankato Foundation Research Grant for this project. Ben is also active on campus having been a member and president of the local ACM chapter, a participant in intramural sports, a member of the lacrosse club team, and a presenter at several on campus presentations. He is currently completing his second internship with IBM, and will be graduating in the fall 2006. After graduation he plans on either attending graduate school or beginning his career in the fields of bioinformatics or software engineering. His hobbies include playing sports, working out, spending time with family and friends, hunting, and anything within the field of computer science.

Faculty Advisor's Biography

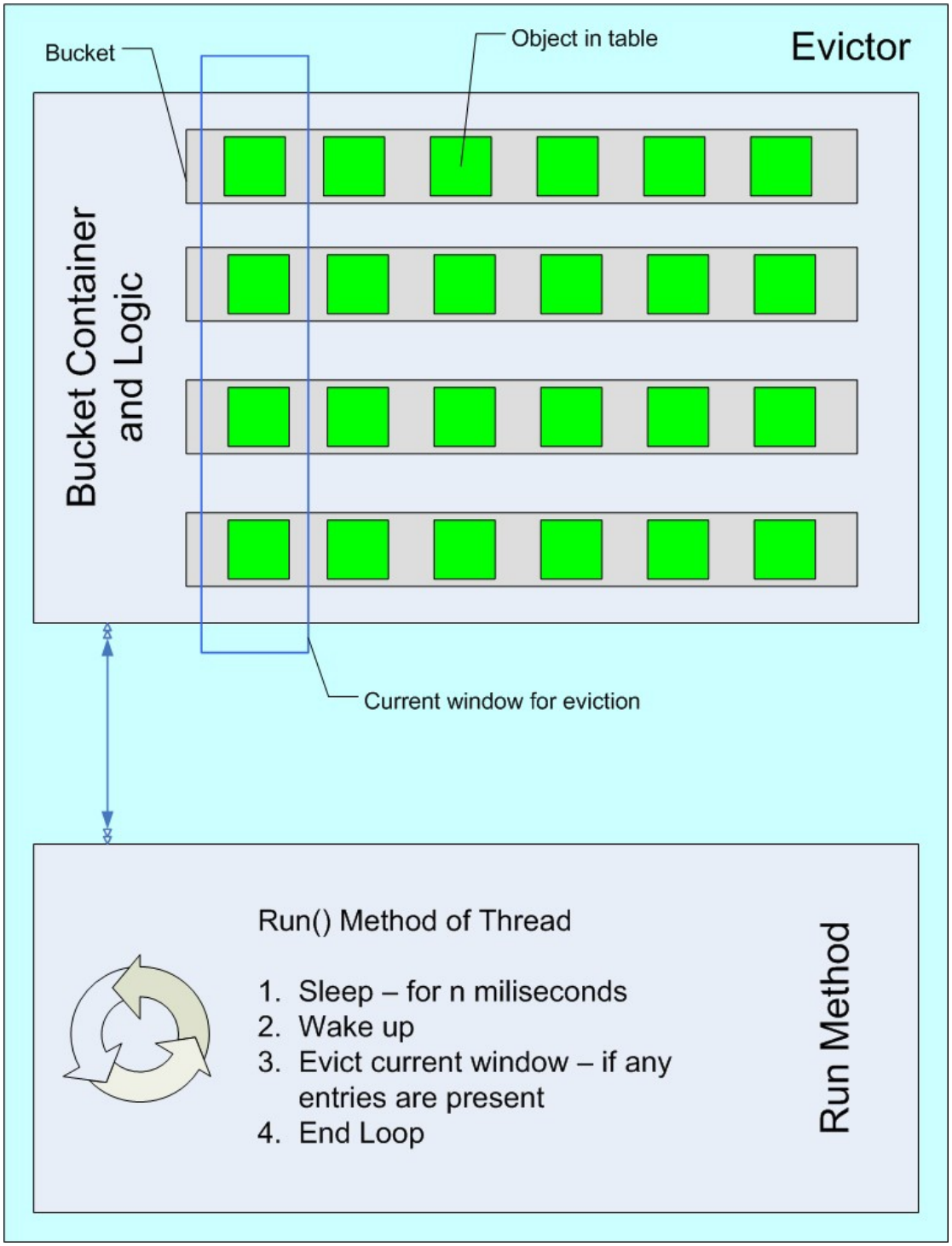
Dr. Ann Quade is a professor in the Department of Computer and Information Sciences at Minnesota State University, Mankato (MSU). She received her PhD from the University of Minnesota.

Throughout her career, she has published research in several areas related to computer science education including: attracting and retaining women in computer science; classroom models that promote undergraduate research; assessing the merits of student online notetaking; the syntax and semantics of learning object meta-data; and the development and assessment of active, project-based hybrid online courses.

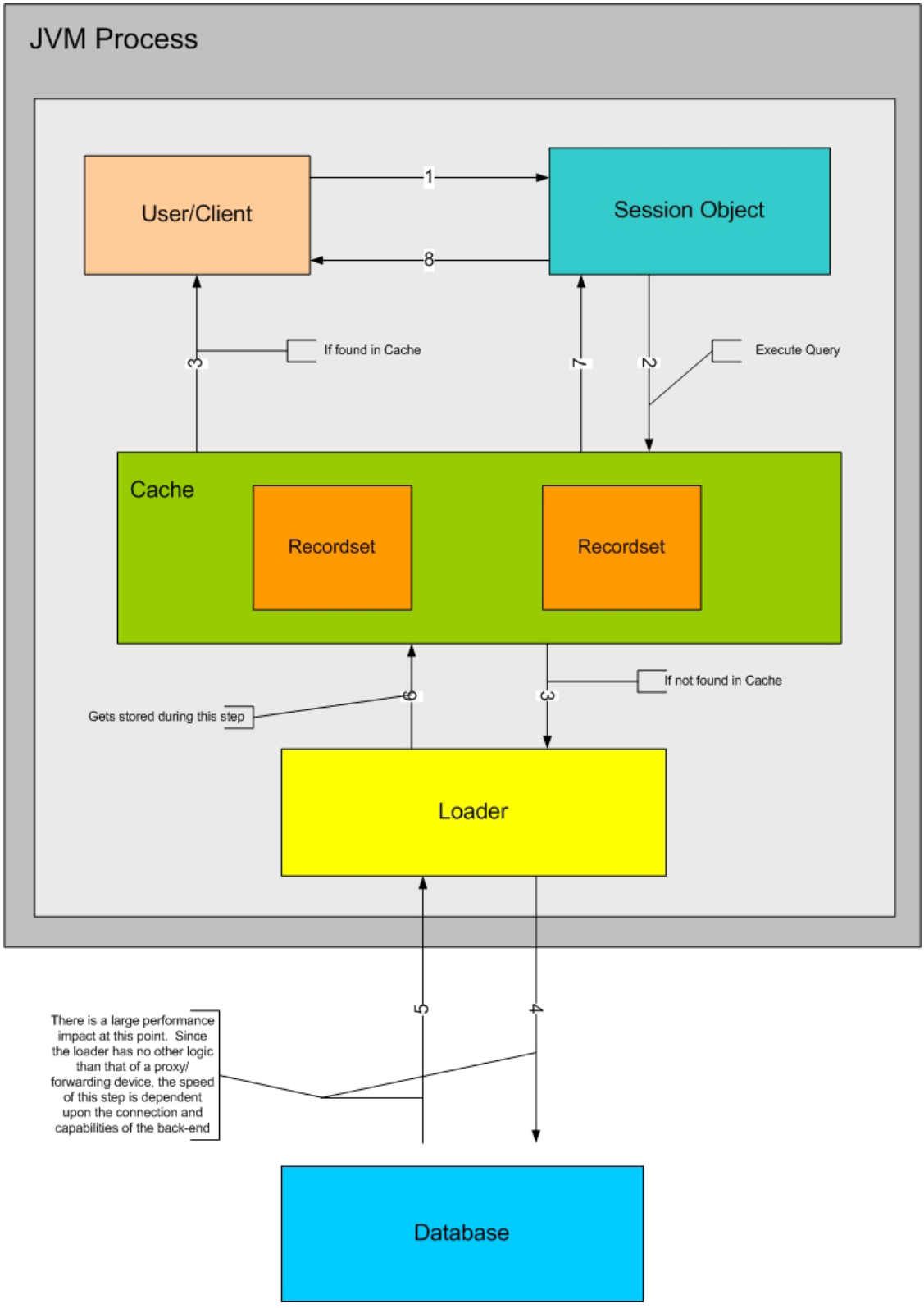
Since becoming a MSU faculty member in 1984, Dr. Quade has represented her University through participation in: numerous national and international conferences; the Computing Research Association mentoring program; and the 2002 International Grace Hopper Celebration of Women in Computing Conference steering committee. In 1998, she presented both oral and written testimony before the U.S. House of Representatives, Committee on Science in support of H. R. 3007, The Advancement of Women in Science, Engineering, and Technology Development Act. She also works diligently to build partnerships between industry and education.

At MSU she has been recognized as a: Teaching Scholar; William Flies fellow; recipient of the Minnesota State Student Association Dr. Duane Orr teaching award; and six year member of the MSU Foundation Board of Directors.

Appendix A: The structural and functional layout of the Evictor class.



Appendix B: The structure and flow of loading a query in Phase 1



Appendix C: The structure and flow of loading a query in Phase 3