Fall 2013

Accelerated Data Delivery Architecture

Michael L. Grecol
Georgia Southern University

ACCELERATED DATA DELIVERY ARCHITECTURE

by

MICHAEL GRECOL

(Under the Direction of Vladan Jovanovic)

ABSTRACT

This paper introduces the Accelerated Data Delivery Architecture (ADDA). ADDA establishes a framework to distribute transactional data and control consistency to achieve fast access to data, distributed scalability and non-blocking concurrency control by using a clean declarative interface. It is designed to be used with web-based business applications. This framework uses a combination of traditional Relational Database Management System (RDBMS) combined with a distributed Not Only SQL (NoSQL) database and a browser-based database. It uses a single physical and conceptual database schema designed for a standard RDBMS driven application.

The design allows the architect to assign consistency levels to entities which determine the storage location and query methodology. The implementation of these levels is flexible and requires no database schema changes in order to change the level of an entity. Also, a data leasing system to enforce concurrency control in a non-blocking manner is employed for critical data items. The system also ensures that all data is available for query from the RDBMS server. This means that the system can have the performance advantages of a DDBMS system and the ACID qualities of a single-site RDBMS system without the complex design considerations of traditional DDBMS systems.

ACCELERATED DATA DELIVERY ARCHITECTURE

by

MICHAEL GRECOL

B.S. Bellevue University, 2009

A Thesis Submitted to the Graduate Faculty of Georgia Southern University

in Partial Fulfilment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

2013

ACCELERATED DATA DELIVERY ARCHITECTURE

by

MICHAEL GRECOL

Major Professor:	Vladan Jovanovic
Committee:	Robert Cook
	James Harris

Electronic Version Approved:
Fall 2013

ACKNOWLEGEMENTS

I would like to thank Anggita Prawiranata for her on-going support, encouragement and editing advise which was instrumental to the completion of this research. In addition, my deepest thanks go out to Dr. Vladan Jovanovic. He never failed to respond to any request quickly and his encouragement, guidance and wisdom was essential to my successful completion of this master's degree.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This paper presents a single solution to the shortcomings of business applications in the form of an application architecture that improves data access. This solution, named "Accelerated Data Delivery Architecture" is a set of standards and components which employ a Hybrid RDBMS and NOSQL system to reduce RDBMS load, make applications more responsive, better control concurrent access and tailor data access to the individual user's needs based on their business responsibility.

In Chapter 2, relevant background information is introduced which will allow a more detailed understanding of the problems. Chapter 3 presents the current state of the art of the areas which the system improves. Chapter 4 introduces these problems in more detail. Chapter 5 explains the architecture, how it works and test data which proves it effectiveness in all of these areas. In Chapter 6, conclusions are made based on the details presented in Chapter 5.

Chapter 2

Background

Business applications are computer programs used by business users to help them perform business functions. The idea behind business applications was to maximize profits by cutting costs associated with repetitive labor intensive operations which can be done proficiently with a computer. This concept was a success because a single computer can replace hundreds of workers doing repetitive tasks. In the early days of computing, business applications took the form of mainframes and dumb terminals. In this architecture all of the processing was done on one centralized mainframe while each terminal served as merely input/output devices. The weakness of this architecture was that the terminals did not allow users to do any kind of data processing other than what was programmed on the mainframe which means that the mainframe controlled all parts of the application from database connectivity to user presentation.

As technologies improved, personal computers became available. The mainframe architecture was replaced with Client-Server technology. This took the form

of thick client applications on the PC's connected directly to database servers. It was a great leap forward as users were able to use their computers to process data locally and still had access to a centralized system. Now, presentation and local processing was done on the user's PC while the data storage took place on the database server.

Client Server technology had severe limitations in terms of scalability and compatibility. The database server's can only handle so many connections from clients before being overloaded. Additionally, these systems were plagued with compatibility problems as any change in programming meant that every client would have to have their application upgraded. This led to undesirable outcomes as clients accessed the same database with different versions of the application.

After the advent of the Internet, which brought the availability of web browsers, business applications took the form of internal web applications using a 3-tier architecture, presentation, business and data tiers[16, 20]. This solved the problems of scalability and compatibility; however, web browsers ability to interact with users was limited. With the advent of Web 2.0 Rich Internet Applications (RIA) became possible which allowed web applications to behave more like desktop applications [17]. These improvements were used in business applications to improve the efficiency in which users interacted with computers.

Throughout the these revolutions, the Relational Database Management System (RDBMS) has been the mainstay of data storage [35]. This is mainly due to its qualities which provide Atomicity, Concurrency, Isolation and Durability (ACID). This means that the RDBMS can be used with any application and guarantees are in place to ensure the accuracy of data. In order to attain improvements in scala-

bility beyond the solitary RDBMS system, Distributed Database Management Systems (DDBMS) have to be employed. The DDBMS system is a database system which can store data across more than one computer [37]. The downside to this is increased complexity and high costs.

Because web applications are based on stateless HTTP protocol, it is dangerous to open a connection to the database longer than a single request [10]. To overcome this, modern web applications only make database connections long enough to retrieve data. Once the data is retrieved, the connection is closed. This is because a user can log off without notice leaving transactions open indefinitely.

Chapter 3

State of the Art in Enterprise Applications

3.1 Database Technology

3.1.1 Relational Database Management Systems

In 1970 E.F. Codd wrote the paper "A Relational Model of Data for Large Shared Data Banks" [14]. His paper proposed storing data utilizing the mathematical concept of relations. This was the birth of the modern Relational Database Management System (RDBMS). His paper addressed a successful storage paradigm; however, modern RDBMS's offer many more services beyond data storage and retrieval. These services include a system catalog, transaction support, concurrency control, recovery services, authorization services, communication services and integrity services among others.

The modern RDBMS exhibits behaviors of Atomicity, Consistency, Isolation

and Durability (ACID) [36, 22, 25]. Each one of these properties is essential to successful implementation of business applications. Below each one of these properties will be explained in more detail.

Atomicity refers to the concept of a transaction. Either all operations within a transaction will complete or none of them will. This is referred to as commit or rollback state of a transaction. For example, if one operation in the transaction experiences an unexpected failure, the transaction may be rolled back. This means that changes in state which occurred before the failure will be reversed. Conversely, if all operations succeed, the transaction may be committed. This means that all of the changes in state that were part of the transaction will be permanent.

Consistency is a guarantee that the database will be in a consistent state. When a transaction begins, data consistency rules will be satisfied and when the transaction ends, data will also meet the databases consistency rules.

Isolation guarantees that the transaction will behave as if it is the only operation being performed. This means that transactions cannot interfere with the changes being made to data items in other transactions.

Durability is the property that guarantees that when a transaction completes, the operations in the transaction cannot be reversed. This property also refers to the database being able to withstand failures and still retain the committed data.

In order for an RDBMS to have acceptable performance, concurrent transactions must be allowed to process. The problem is that concurrent transactions can have undesirable results if not controlled properly. Kim, et al. has provided a taxonomy of dirty data which expands on some of these dangers [30]. Improper transaction management can result in four types of dirty data:

1. Lost update

2. Dirty read

3. Unrepeatable read

4. Lost transactions

A lost update is when two transactions update the same piece of data one after another known as a ww conflict. The result is that the first update becomes overwritten by the second update. A dirty read happens when transaction one reads data that was changed but not committed by transaction two known as a wr conflict. If transaction two later rolls back, then transaction one's updated data was based on inaccurate values. An unrepeatable read is when transaction two writes data after transaction one reads it also known as a rw conflict. If transaction one later updates the data it will be based on inaccurate values [37]. A lost transaction is when a system failure happens before a transaction is committed. The durable property guarantees that once a database transaction is committed it becomes permanent, but it is possible that transactions are lost before they commit [30].

In the forty plus years of RDBMS research, many transaction models have been proposed. Dr. C. Mohan has surveyed some of these advanced transaction models. The most popular of these transaction models is the traditional ACID model which is based on serializablity [34, 37]. Although there is a multitude of transaction models, most of them are based on long-lived transactions outside the scope of the database server. Microsoft allows the administrator to choose either optimistic or pessimistic concurrency control in their SQL Server 2008 R2 product.

Additionally, the server allows for setting of various isolation levels from read uncommited which allows for dirty reads to serializable which ensures no transactional interference. The read uncommitted setting is the fastest as it allows for full parallel operation. On the other hand, serializable transactions must collect and order transactions so as to avoid any interference.

3.1.2 Distributed Database Management Systems

Distributed Database Management Systems (DDBMS) are defined as database management systems which can store data across more than one computer [37]. DDBMS's may consist of a collection of RDBMS's or other DBMS's known as clusters.

DDBMS's usually have the property of location transparency which means that the user who is executing queries does not need to know the location of the data. The system acts as if it is a single logical system. In a DDBMS, tables may be partitioned or replicated. Tables may be partitioned horizontally, vertically or both depending on design. The design of the system is a very complex operation as it's aim is to ensure high read performance by strategically locating data where it is needed.

Traditional DDBMS's may offer the same ACID guarantees of single-site RDBMS's. The most popular technique to maintain consistency is two phase locking (2PL) [18]. In this approach, there are two phases, the growing phase and shrinking phase. In the growing phase, the transaction acquires locks but does not release any lock until all of the its requested locks have been granted. After all of the locks have been granted, the shrinking phase will begin to release locks. In order for this

to work in a distributed environment, a common interface has to be put into place.

The eXtended Architecture (XA) published by X/Open group is a standard for global transactions that has been implemented by most major commercial database systems [2]. This standard defines the three components of distributed transaction processing as:

1. The application program which defines the transaction boundaries.
2. Resource managers which are databases or other data sources.
3. Transaction manager which monitors transactions and takes responsibility for transaction completion and failure.

In distributed transactions with 2PL, a data item may be locked while the transaction has already moved to another site to lock additional items needed for the transaction.

Distributed deadlock management can take the form of deadlock prevention, deadlock avoidance and deadlock detection and removal [37]. Although a multitude of deadlock management algorithms has been proposed, mainly detection and removal has been implemented in commercial software. Microsoft's SQL Server 2008 R2 and Oracle 11g both use deadlock detection and removal to handle deadlocks as do many DDBMS implementations [8, 3, 37]. This scheme requires that a deadlock detector is running on at least one machine to poll all of the distributed servers and detect possible deadlocks.

One special feature of DDBMS's is replication. Replication is used in DDBMS's to increase performance by allowing maximum parallelism on data read opera-

tions by locating the same data on several different sites at the same time[37]. Having many locations of the same data is a risk to the ACID guarantees of the data. In order to ensure the ACID guarantees, all replicated sites must be updated on every update before the update is committed. Generally, mechanisms like majority voting or circulating tokens are used to avoid deadlocks and ensure synchronized updates of all replicas [37].

3.1.3 Not Only SQL

Modern application environments require very responsive database systems to drive their performance. After many years of relational RDBMS dominance, the state of the art in performance is changing toward distributed database systems. Dr. Eric Brewer has proposed that distributed systems can at most have only two of the three properties of Consistency, Availability and Partition tolerance (CAP) this is known as CAP theorem [12]. This was seen as a loosening of the restrictions of ACID guarantees and has led to the develop of Not Only SQL (NoSQL) database systems. These are fast, highly distributed systems designed for quick access to data.

NoSQL systems subscribe to the Basically Available Soft-state Eventual consistency (BASE) concept of consistency control [36]. BASE-oriented consistency control accepts out of date data in exchange for the advantages of distributing data. This data will eventually be consistent according to the BASE model. Many different types of NoSQL systems have been implemented. NoSQL databases offer much higher throughput than traditional RDBMS's [39]. Additionally, these

databases are designed to run well on less expensive hardware [39]. Cattell classifies these new database systems as three types, namely [13]:

1. Key-value stores
2. Document stores
3. Extensible record stores

Key-value stores are the simplest type of NoSQL database. These databases use an indexed key to refer to the location of a piece of data [13]. Document stores are the next level of complexity which store indexed documents. These systems can store many types of documents, can have multiple indices's and have more robust query functionality than key-value stores. Extensible data stores, based on Google's Big Table, utilize a row/column data store with the ability to store rows and columns over different locations [13].

MongoDB is an open source NoSQL document data store. This database does not respond to SQL queries but offers much of the functionality of SQL through its robust query interface [4]. Instead of using the SQL language, MongoDB uses a combination of query commands and Binary Java Script Object Notation (BSON) [1] which is a schema-less compact serialized interchange format. This allows the database to have rich query functionality. Additionally, MongoDB offers aggregation, sharding and replication services[4].

3.1.4 Hybrid Database Systems

Hybrid databases is a concept that may solve the combined needs of performance and consistency. Current research on hybrid database systems is mainly limited to a mixture of in-memory to RDBMS database systems such as SAP's HANA database [24, 19, 38]. Hybrid Databases have existed since the mid 1990's with WebDNA's hybrid in-memory - RDBMS solution [28]. Additionally, hybrid NoSQL - RDBMS systems are beginning to emerge; which currently are limited to cloud services or experimental designs [9].

3.2 Web Application Technology

Web-based business applications are often arranged in three-tiers [16, 20]. The first tier is the presentation tier, the second is the business logic tier and the third is the data tier. Each tier represents a physical site. For example, the presentation tier is the web browser, while the business logic tier is the application server and the data tier is the RDBMS. This allows for the application to be distributed, thus optimized performance.

The connection between the presentation tier and the business logic tier is HTTP protocol run over internal or external networks. This allows for reliable communication of content to and from the browser. Communication to and from the data tier usually consists of some form of TCP/IP communication.

A leading design architecture is N-Layer architecture. In this architectural style, layers are used as logical divisors of components [16]. Each layer has separate responsibilities and changes to one part should minimally impact another part.

This promotes a flexible environment where a layer's physical location can change with less effort and maintainability is improved[16].

N-Layered architecture consists of several layers such as presentation, application, business, infrastructure and data persistence layers. Each layer has a specific function to perform. Each layer works together beginning and terminating at the presentation layer. In order to process a transaction each layer has to be transversed to ensure predictable execution paths which aid in design, implementation and maintenance.

3.2.1 Object-Relational Mapping

Object Relational Mapping (ORM) systems are application components which adapt relational database objects with programming language objects. These systems provide methods in which objects may be read or written to the database system through manipulation of in-memory objects. Two popular ORM systems are Java Persistence Architecture and ADO.NET Entity Framework [10]. In modern transactional applications, ORM is the standard for database retrieval and manipulation. Modern application architectures are designed around using this as the primary data access method.

ORM systems are designed to make the application RDBMS vendor transparent to the developer. This gives the enterprise the ability to change RDBMS vendors of the application. In other words, if a system is built with ORM technology, one may migrate the application to a different database vendor with very little programming changes.

3.2.2 Browser Technology

The standard display component in modern applications is the web-browser. Web Browser technology has traditionally been based on Hyper Text Markup Language (HTML) and Javascript. HTML has undergone five major version changes since its inception. The current standard, HTML5, allows for a number of new features that are helping to drive highly responsive application systems [7]. Firstly, HTML5 improves interactive behavior through native support of new media types. Additionally, HTML5 sets out a standard for a browser-based database system known as indexeddb. Indexeddb is a NoSQL key-value datastore which allows the browser to persist data in the browser environment. The browser can execute queries and update data through Javascript running in the browser.

Additionally, the V8 Javascript engine which is now available in many browsers offers performance advantages over older Javascript engines [6]. HTML5 and V8 represent significant advancements in browser technology. They have the potential to transform the browser from a display platform to a computing platform.

3.3 Prefetching Technology

Prefetching has been used to decrease latency time in a multitude of environments. In a study of prefetching low-level instructions, [32] found that the overhead of prefetching instructions was minimal compared to the gain. Similarly to our application, operating systems also use predictive prefetching to make the system more responsive. Microsoft windows Vista and above use a system called SuperFetch which analyzes memory usage patterns and preemptively loads memory

based on past history [5]. This will be accomplished through the use of analytics to predict user actions before they happen. Prefetching methods may be classified in dimensions of either prediction engine used and granularity of prediction. There are four known types of prediction engines: strategy-based, structure-based, hint-based and context-model based [21]. Strategy-based methods used explicitly programmed strategies to determine what to prefetch. This method prefetches based on explicitly defined groups contexts. Structure-based prefetching uses relationships or other structural information to determine what to prefetch. Hint-based algorithms use hints provided by the application like access patterns. Context-model-based prefetching uses preceding events to predict the next event[27].

Khemmarat et al. proposed context-model-based prefetching to decrease the latency of retrieving user uploaded videos[29]. They did this by identifying which video should be prefetched and how much of that video. In their case, the related video list and search result list was used to predict the user's next action.

Han et al. has proposed using type-level access patterns to prefetch data in a ORDBMS [26]. This model uses the concept of prefetching objects as opposed to disk pages based on navigational paths of objects. This is accomplished by using type-level access locality and type-level path to predict the next object to be retrieved. In other words, lists or recursive data elements embedded into ORDBMS objects are prefetched based on predictions.

He et al. proposes a path and cache conscious model called PCCP [27]. This model uses both database page and object access to make predictions and prefetch at the database page level. Similarly to the type-level path, statistics are used to keep statistics on object access which is used to predict the next database page to

be retrieved. It introduces the concept of cache consciousness where if a certain page has the probability of already being in memory it is not prefetched. This allows it to prefetch a database page far in advance based on a small navigational sample. It uses feature point selection to determine the drivers of the navigational path. In this prefetching scheme the calculated navigational path is used to predict and prefetch database pages. This method uses training data to calculate the navigational paths in advance. The term cache concious refers to using historical training data to determine which pages will always be resident in the cache or not. These pages are marked as resident or non-resident and allow the prefetching system prefetch data earlier.

Chapter 4

Problems Current Technology

4.1 Problems with Database Technology

4.1.1 Relational Database Management System

As mentioned in 3.1.1, RDBMS systems are sensitive to concurrent transactions. To maintain ACID properties, transactions must be executed in serialized order. Industrial strength RDBMS systems allow for setting the isolation level which is defined as the degree in which a transaction may interact with other transactions [15]. At one extreme, isolation levels allow a transaction to read uncommitted data. This gives the system high performance; however, it introduces the dangers of dirty data discussed in 3.1.1. At the other extreme, transactions can only execute in serializable order. Serializability is ordering transactions so they are equivalent to a serial schedule[37]. This is required to maintain the ACID properties discussed in 3.1.1. The weakness in this isolation level is that resources are consumed to check serializability and the it limits the amount of parallel operation that the server can

perform [3, 37].

In addition to this, the cost of licensing a RDBMS system is very high. This causes the enterprise to load multiple applications on a single RDBMS server to maximize utilization of that resource. The effect of this loading is that the performance of each application decreases as the RDBMS becomes loaded down. This introduces an enigma for architects, how to maximize the RDBMS utilization and still have responsive applications.

4.1.2 Distributed Database Management

Section 3.1.2 introduced DDBMS systems. One way to increase the performance of an RDBMS is the distribute data to a cluster of RDBMS's into a DDBMS. DDBMS systems require a number of components to ensure ACID compliance. Each of these components uses resources and some require a separate server. This increases the complexity of a system. Assuming a DDBMS may be a series system in which data objects only exist at one location at a time, reliability of a system can be calculated as seen in Equation 4.1 [11]. Let Q_S be the unreliability of the system while R_i be the reliability of a component in that system. It is easy to see that having more components make s system more unreliable. To counteract this issue, components will have to be put in parallel as redundancy [11]. This means that hardware must be purchased, configured and maintained.

$$Q_S = 1 - \prod_{i=1}^{n} R_i \tag{4.1}$$

In addition to the basic problem of reliability, new dangers of deadlocks are introduced in DDBMS systems. As transactions are executed across many sites, it is possible that a transaction is open on one site while the transaction has moved to another site to complete work. This greatly increases the likelihood of a deadlock occurring. Additionally, if a site goes down while a distributed transaction is in process, it can block other processes from resources. Therefore, deadlock detection systems must be put into place as seen in 3.1.2.

The downside to all of this increased complexity is increased cost and risk. As the complexity increases, so does the cost of equipment, maintenance and licensing.

4.1.3 Not Only SQL

NoSQL systems as described in 3.1.3 offer many performance advantages of the traditionally DDBMS systems as described in 3.1.2. However, the key driver of this increased performance is the fact that these systems do not have traditional ACID properties. This is a problem for most business application as a key requirement of the DBMS are the ACID properties themselves.

NoSQL databases do not offer the full functionality of a RDBMS database. For example, MongoDb is a BSON document datastore which means that data is stored and processed as BSON documents, not relations [4]. Therefore, joins are not natively supported. Unlike RDBMS tuples, a BSON document may include objects and arrays [1]. This means that NoSQL databases violate Codd's 1st Normal Form (1NF) [14] because an attribute can contain more than one single value.

In NoSQL databases, an attribute can contain an array or an object or even an array of objects. This is useful for high performance as expensive joins are avoided and related data may be stored in a single document. While not adhering to 1NF increases flexibility and speed, it opens up the possibility of update anomalies. Therefore, to update related data, one would have to find all objects which include the subject data and update them as well. On the other side, data in Cod's 3rd Normal Form, the standard for relational data, suffers from difficulties in representing data accurate to transactional time as referenced in [31]. NoSQL databases do not suffer from that because updated reference data is only updated at a point of time forward unless older objects are explicitly modified.

This is very significant because in order to build an application to be used with a NoSQL database system, one must consider these design aspects at the beginning of the project and have a solid commitment to this technology. Each component would have to be built around violating 1NF which means that the DBMS cannot be changed to a RDBMS without a complete system redesign.

4.1.4 Hybrid Databases

Section 3.1.4 introduced several type of hybrid database systems. These databases systems have potential to solve the RDBMS performance problems; however these are not practical for business appplications. For example, in-memory databases require specialized hardware which may not be an acceptable cost for the enterprise. Additionally, hybrid NoSQL - RDBMS systems are currently are limited to cloud services or experimental designs, not hybrid architectures to be used with

industrial strength RDBMS systems like SQL Server or Oracle database [9]. Enterprises require proven technologies to be used with critical data. Risking critical data with experimental technology or DBMS's without a demonstrated history of reliability will be unacceptable to the design requirements of many business applications. Furthermore, maintaining experimental systems in a time-critical manner will not be possible as IT staff would need experience in such systems to keep them running well.

4.2 Problems With Web Application Technology

As mentioned in 3.2, web applications run off a disconnected database model. This means a connection is made to the database long enough to retrieve data. Once the data is retrieved, the connection is closed. With this model, even with ACID guarantees at the database level, there is no guarantee that the data is current once it is loaded from the application. The traditional method of ensuring data is valid is to leave a transaction open when the data is retrieved, that is not an option in modern applications[10]. Transactions take up valuable database resources this would severely limit the number of users in the system. Additionally, the stateless nature of HTTP means that users could disconnect from the application without notice leaving transactions open indefinitely. Since concurrency is only controlled within the database itself and not at the application level, applications rely on frequent polling to detect changes in data. This leads to unnecessary database loading. Lin, et al. proposed database replication to move data closer to the user and reduce loading of the main RDBMS [33]. This greatly increases the expense database li-

censing.

N-Layer design which was introduced in 3.2 meets the needs for business applications that can benefit from distributed components. However, no standardized way of using more than one database system has been proposed as a part of N-Layer design.

Applications use the database for a varied number of requirements such as critical business data, reference data and application configuration settings. Current designs are systems-centric meaning they typically look inward at the system design and components. This being the case, architectures of the day with a system-centric view do not explicitly define how storage should be handled for the varied uses of the data. The next level of design methodology should look beyond the system to the ontological aspects of the data and make design decisions based on the true meaning, requirements and value of the data.

4.2.1 Object Relational Mapping

The problem with ORM systems as described in 3.2.1 is the loss of control on the exact data manipulation statements that are sent to the DBMS. This is particularly problematic with update transactions. Depending on the implementation of the ORM system, only changed attributes may be updated or entire objects. This loss of update statement control coupled with the problem of concurrency control mentioned in 4.2 may lead to concurrently updated data being lost. The default concurrency control for ORM systems is optimistic concurrency in Last Writer Wins (LWW) fashion [40]. This poses a problem when two updates are executed on the

same object. Due to the fact that ORM frameworks may update an entire object at once, there is a danger of loosing updated attributes. This happens when updates are performed on two different attributes of the same parent object. Although the two attributes are not in conflict, but their parent object are subject to WW conflicts defined in 3.1.1. This results in non-conflicting data from first transaction being lost.

The lack of control may lead to performance problems as well. Since the developer has no control over the actual language sent to the RDBMS, the ORM system may not send fully optimized queries. Developers may be able to develop advanced query methods which are intended to increase performance; however, these may be specifically written to a vendor-specific RDBMS which nullifies the ORM's advantage of database vendor transparency mentioned in 3.2.1.

4.2.2 Browser Technology

Despite the advances in browser technology, the web browser still works on a stateless protocol. This, coupled with the disconnected database model mentioned in 4.2, browser-based applications subject data to concurrency anomalies.

Additionally, browser-based database systems use has not been standardized in applications. Due to the larger problem of system-centric design against ontological value of data seen in 4.2, architects are ill prepared to use web browsers in a standardized way. This has caused the browser-based database system to be under utilized in practice.

4.3 Problems with Prefetching

The drawback to type-level prefetching mentioned in 3.3 is that prefetching is limited to type-level access locality which may not be the user's next activity. If the next action does not have type-level access locality, the system will not prefetch accurately. Also, since it is not user conscious, it is not able to discern requests from multiple users.

He et al.'s proposal is designed to work well when there are repetitive patterns of database access within an application. It may not be effective for multiple users requesting different pieces of data concurrently. Additionally, when users retrieve a new object on every request, this model may not prefetch these new objects.

Enterprise web applications differ from consumer internet applications as their usage is based on individual's work process. Meaning, that each person concurrently working in the system may be viewing and updating different parts of the database based on area of responsibility. For example, some users manage data by account and others by time period. This paper's contribution is geared specifically towards enterprise web applications; therefore, the examples and scientific proof given applies to specifically to these applications.

In this section we will present an example of an enterprise web application. A simplified data model for this application can be seen in Figure 4.1. Let's assume, for the sake of this example, that there are two concurrent users at the time of the analysis. User A is a shipping department user who's function is to view the orders, prepare packages for shipping and enter shipping information into the system. User B is an accounting user who's function is to update the payment status

of statements and invoices. To illustrate the efficacy of each prefetching concept lets assume that each database page holds five rows of data and the pages hold rows sequentially in order of the id column. An example of the user's actions in the system are listed in Table 4.1. Each one of these user actions are accompanied by queries to several tables in the database and rendering the HTML response to the browser. Using this scenario, we can project the efficacy of the two prefetching schemas.

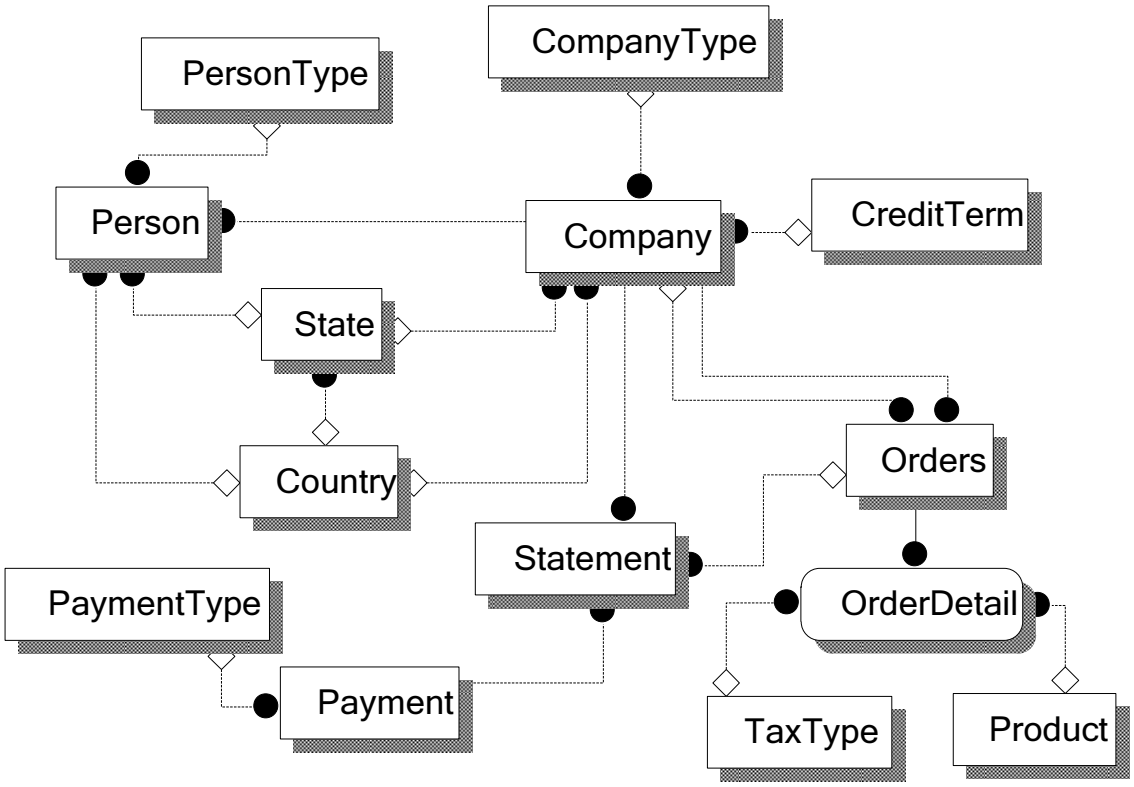

Figure 4.1: Data model of test system.

Next is an analysis of Han et al.'s Type-Level access pattern concept using the user activity provided in Table 4.1. Additionally, each user screen is associated

Table 4.1: Sequence of actions for example application.

Transaction	**User A Action**	**User B Action**
1	Login	
2		Login
3	Load Portal Screen	
4		Load Portal Screen
5	Open Accounting List	
6		Open Shipping List
7	Open Statement #10	
8		Open Order #100
9	Update Statement #10	
10		Open Order #101
11	Open Accounting List	
12		Open Shipping List
13	Open Statement #18	
14		Open Order #102

with one or more database entities from Figure 4.1 these associations are listed in Table 4.2. For this analysis, lets assume that the Type-Level Paths have already been generated. A table of transactions and the prefetching algorithms performance is listed in Table 4.3.

Starting at transation 1, the navigational root is the Login screen. When this is accessed, the algorithm will follow the navigational path, but there is no person or company object being queried until the person actually logs on. Once the user logs on, the user's company will be put into cache. Then, the portal screen is opened on transactions 3 and 4. The portal page contains links to work lists and news. The system has not loaded any objects from the ORDBMS, only statistics and news; therefore, no prefetching activities have happened. On transaction numbers 5 and 11 the accounting list is loaded from the database. When these screens are loaded, the prefetching algorithm loads all of the associated payments, invoices and com-

panies related to the records listed on the screen. Concurrently, the shipping list is loaded from the database on transaction 6 by User B. This causes all of the shipments, packages and carriers to be prefetched into the cache. This prefetch does not cache any data because at this time, there are no shipments, packages or carriers attached to the order. The same pattern repeats itself on subsequent transactions. Therefore, we can conclude that this algorithm will not prefetch any of the worklists nor the objects within the worklists; however, the data related to the those objects will be fetched. This should improve the application slightly; however, since an enterprise application is based heavily on worklists, the improvement is less than optimal. Additionally, no facility is made with this prefetching algorithm concerning concurrency and outdated data in the cache. This violates the ACID contract as blind write (WW) inconsistencies may occur as defined in 3.1.1.

Table 4.2: Screen to database table associations for test application.

Screen Name	**Database Tables**
Login	Person, Company
Portal	Person, Company, Statements, Orders
Accounting List	Statements
Statement Record	Statements, Payments, Invoice, Company
Shipping List	Orders, Shipment
Order Record	Orders, Shipment, Packages, Carriers

Applying the concepts of PCCP to the same example, the predicted prefetch results were determined in Table 4.4 and a reference of database pages is provided for in Table 4.5. The PCCP algorithm assumes that carriers, person and company are always in memory; therefore, they are marked as resident and not prefetched. We also assume that the PCCP algorithm prefetches pages that relate

Table 4.3: Prefetch results for Type-Level Prefetching.

Sequence	**Prefetch Action**	**Database Action**
1	+Company	Q:Person Object.
2	+Company	Q:Person Object.
3	∅	Q:News
4	∅	Q:News
5	+Payments, Invoices, Companies	Q:Statements
6	+Shipment, Packages, Carriers (NULL)	Q:Orders
7	Q:Payments, Invoices, Companies	Q:Statement
8	Q:Shipments, Packages, Carriers	Q:Order
9	∅	U:Statement
10	Q:Shipments, Packages, Carriers	Q:Order
11	+Payments, Invoices, Companies	Q:Statements
12	+Shipment, Packages, Carriers (NULL)	Q:Orders
13	Q:Payments, Invoices, Companies	Q:Statement
14	Q:Shipments, Packages, Carriers	Q:Order

to the id's that were retrieved in the access path. Beginning with transactions 1, 2, 3 and 4, the PCCP algorithm does not prefetch any data because the algorithm cannot determine the user's access path because different users access different parts of the program depending on their job function. When user A's action causes the database to query statements for the accounting list, the algorithm sees the combination of orders→statements and prefetches the payments entity because previous users fetched payments directly after looking at statements. As seen in transaction 5, Page 20 of the payments entity is put into the cache. On transaction 6, user B opens the shipment list, the PCCP algorithm sees the sequence of statements→orders, shipments, from the combination of transaction 5 and transaction 6. Since the combination of statements→orders usually results in a query of invoices, the PCCP algorithm prefetches page 20 of invoices corresponding to records 96-100. In transaction 7, user A opens the statement 10, the PCCP algo-

rithm sees the combination of orders, shipment→statements, payments, invoice and prefetches page 2 and 20 of packages. In Transaction 8, user B opens order 100, the PCCP algorithm sees the navigation of statement, payment, invoice→orders, shipment, packages, carriers. This navigational path does not have any training data, therefore it does not prefetch any data. The packages entity for order 100 is already in the cache, so the algorithm uses the cache data for this. Transaction 9 is an update, so no data is prefetched. In transaction 10, user B opens order 101 which is not in the cache and it has no navigational path to evaluate since the last statement was a prefetch; therefore, no prefetch operations take place. In Transaction 11, order 101 is opened, the algorithm sees the combination of orders, shipments, packages→statements the PCCP algorithm prefetches pages 20 of invoices because the it correlates to shipment 101 and the accounting list. Transaction 12 is essentially the same as transaction 6; therefore, the same result applies. Transaction 13 loads pages 4 and 21 of the packages entity because statement 18 exists on page 4 and the latest shipping list now is page 21. Transaction 14 finds the packages entity that it needs is in the cache while the remainder of items is queried from the database.

These prefetching algorithms are effective for many systems in which access patterns are predictable; however, in an enterprise's operational systems where each user's access patterns are unique they cannot predict operations at the database or application level. Since several user's actions are interleaved prediction without user awareness can result in less than optimal prefetching. Additionally, prefetching schemes have a very limited time frame in which to calculate predictions and retrieve data. Additionally, memory resources are always limited. Using mem-

Table 4.4: Prefetch results of PCCP algorithm.

Sequence	**Prefetch Action**	**Database Action**
1	∅	Q:Person Object.
2	∅	Q:Person Object.
3	∅	Q:News
4	∅	Q:News
5	+Payments-Page20	Q:Statements
6	+Invoices-Page20	Q:Orders, Shipment
7	+Packages-Page2,20	Q:Statement,Payments, Invoices, Companies
8	Q:Packages	Q: Carriers, Order, Shipments
9	∅	U:Statement
10	∅	Q:Order,Shipments, Packages, Carriers
11	+Invoices-Page20	Q:Statements
12	+Invoices-Page20	Q:Orders, Shipment
13	+Packages-Page4,21	Q:Statement,Payments, Invoices, Companies
14	Q:Packages	Q:Order,Shipments,Carriers

ory as a cache may lead to wasted resources or cache misses meaning an object is expelled from the cache just before it is actually needed.

Table 4.5: Database pages and their corresponding records.

Database Page	**Records**
Page1	1-5
Page2	6-10
Page3	11-15
Page4	16-20
Page5	21-25
Page6	26-30
Page7	31-35
Page8	36-40
Page9	41-45
Page10	46-50
Page11	51-55
Page12	56-60
Page13	61-65
Page14	66-70
Page15	71-75
Page16	76-80
Page17	81-85
Page18	86-90
Page19	91-95
Page20	96-100
Page21	101-105

Chapter 5

Data Acceleration Architecture

In order to achieve the consistency of a traditional RDBMS coupled with the performance advantages of a NoSQL DBMS, this paper proposes an application architecture as middleware to distribute data between an industrial strength RDBMS and a cutting-edge NoSQL system. To demonstrate the idea (proof of concept) we use Microsoft's SQL Server as our RDBMS which is a proven industrial strength RDBMS solving reliability and maintenance issues in 4.1.4.The system also uses MongoDB as our NoSQL database and the chosen middleware architecture is the J2EE framework.

The ADDA architecture has two main concepts which drive the design, namely the Defined Consistency Levels and Data Leasing System. These two concepts allow for an application to be developed independently as a standard three-tier design as defined in 3.2 and later ported to a distributed data implementation. It gives the architect flexibility to define objects in different ways to meet the requirements of the business thus making design decisions based on the requirements,

meaning and value of data as mentioned in 4.2. These two concepts are explained below.

1. Data Leasing System. For entities which require high consistency level a tight contract between application and a database is needed to ensure that the application always has the most recent data. Having a leasing system which ensures the freshness of the data and manages the access carefully allows reduces polling and increases control of critical data items helping to solve concurrency issued defined in 4.2. This reduces the risk of WR and WW interference mentioned in 3.1.1. Additionally, entities of lower consistency level are routed through the leasing system which manages the storage location and currency of the data.

2. Defined Consistency Levels. Not all entities in a database require stringent ACID requirements. As such, certain entities may be distributed safely. Defining the required consistency levels for entities is the precursor to the architectural framework that this paper proposes. Once this is defined, certain data entities can achieve the performance advantages of distribution based on their consistency level.

5.1 Data Leasing System

The topology of the ADDA system consists of the three tiers much like the standard web application architecture [16, 20]. However the difference lies in the use of several data stores. ADDA utilizes a NoSQL datastore at the application tier which

may also be moved to a forth tier but we leave this for future elaboration. Additionally, it utilizes the browser-based database consistent with the newest browser based technology, HTML5, at the client tier. Figure 5.1 shows the internal components of the ADDA system with the presentation tier at the top, the business logic tier in the middle and the data tier at the bottom.

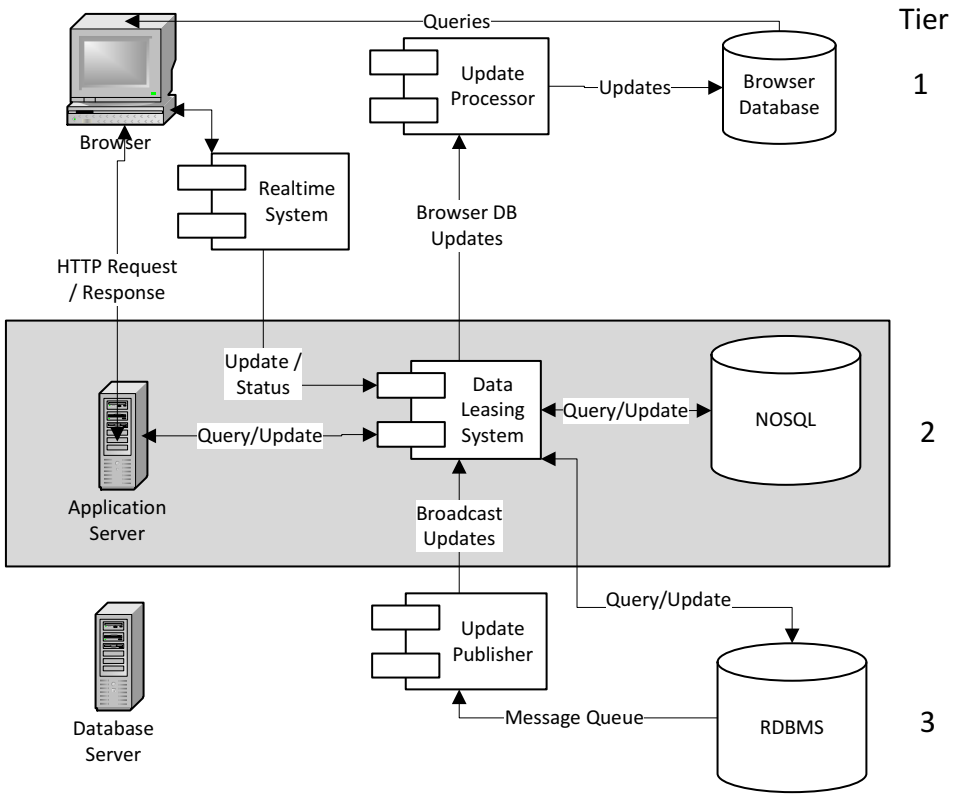

Figure 5.1: Example of ADDA architecture.

The main component in the ADDA system is the data leasing system. It is the central component which includes the logic for all data operations. One function of this component is to route queries and updates to the correct data store. When a query is issued to the data leasing system, it reads the requested item's consistency

level and retrieves the data from the correct data source. If the data source is the indexed db, then the Javascript code to retrieve the data is included in the HTML page which is sent to the browser. Another function of the data leasing system is the update engine.

The data leasing system uses ORM mapping to propagate data across several heterogeneous databases. It uses timestamps to determine if a concurrent update had occurred on the data object. If it hasn't, then the object is updated. If concurrent updates have occurred, the system updates only fields that were changed since the last update. In this way, it avoids overwriting attributes of concurrently updated object as mentioned in 4.2.1. If the system has more than one application server, the system will update in Last Writer Wins (LWW) manner. This is consistent with current methodology concerning optimistic concurrency [40].

Updating the browser database is quite different. The HTML responses sent to the browser include a Javascript component named the update processor. The update processor checks the current status of the browser database and sends the largest timestamp during its normal request to the server. If the data leasing system determines an update is needed, it collects the update and puts JSON data in the clients update queue at the application server. When the client requests another page, the update queue is sent with the response. When that is received by the browser, the update processor updates the associated records in the browser database. In this way, the client never waits for a page to load while the server is processing its browser database update. Instead, it sends the data as a part of the next response. The will present the framework to a standardized architecture to utilize the browser-based database system solving the problems of 4.2.2.

The Data Leasing System also manages subscriptions. Subscriptions are used in the system to provide realtime updates to high consistency level data. This helps to solve web application issues not being able to keep transactions open as dicussed in 4.2. When an object is updated, the update publisher sends a message about the updated objects to the data leasing system. This immediately updates the data at the application server. Therefore, there is no need for the polling of the RDBMS for the current data which helps to reduce RDBMS burden which may cause it to slow down as noted in 4.1.1. In this way, the system can keep an object in memory for fast performance while immediate subscription updates keep it up to date. In order for this to work, the data leasing system works with the realtime system which is a Javacript component which runs on the browser. The realtime system informs the data leasing system when the client is viewing certain objects and updates the screen in real time.

The data leasing system also can lock data objects using block-free locking mechanisms. Current methodology for concurrency control involve locking data objects by blocking access to them. This interrupts the control of a program making it unresponsive. It also introduces the risk of deadlocks. The data leasing system locks the data object by leasing it to a specific user. If that user updates the object, the data update is accepted. If another user attempts to lock the object, it will fail as it is already locked by another user. Additionally, if another user attempts to update the object, the operation returns failed immediately because it is locked by another user. This allows for superior consistency control while allowing the application to handle failed transactions without blocking control. The architect can choose to use a timed retry, to queue their update in the background or to ask

the user to retry. The data leasing system can maintain leases on critical objects by using the update publisher to track when a user has control of an object.

5.2 Defined Consistency Levels

The concept of consistency levels allows for the application to be designed using a single data model. That data model is classified into the consistency levels shown in Table 5.1. In this way, the consistency level will control the storage methodology and location of data.

Table 5.1: Classifications of consistency levels and their meaning.

Consistency Level	**Definition**	**Data Location**	**Timing of refresh**
Passively Consistent (PC)	Object is refreshed based on a milestone.	Browser	Object is checked during login or other milestone and refreshed if needed.
Gradually Consistent (GC)	Object is refreshed in timed intervals.	NoSQL	Next Page Refresh after time expires if object has changed.
Eventually Consistent (EC)	Object is refreshed after it is changed.	NoSQL	Next Page Refresh
Always Consistent (AC)	Object is always queried from the database.	RDBMS	Next Page Refresh
Immediately Consistent (IC)	Object is always queried from the database, and updated in real time.	RDBMS	Realtime

In order to achieve the dynamic hybrid DBMS through defined consistency levels and the data leasing system, data items must be assigned a consistency level. At the lowest form of consistency level, data is allowed to be displayed at a stale

state, preferring fast response time over freshness of data. At the highest consistency level, the data must always reflect the current state of the RDBMS. A listing of the consistency levels and their definitions may be seen in Table 5.1.

5.2.1 Passively Consistent

Passively Consistent objects utilize in-browser databases in a standard way solving the problems seen in 4.2.2. The advantage is speed as requests do not have to go to the server to be processed as seen in Figure 5.2.

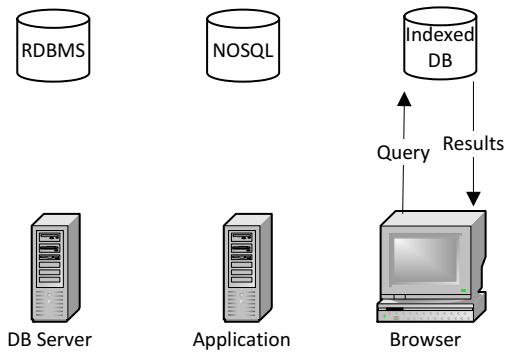

Figure 5.2: Passively consistent query.

Passively consistent data is synchronized when the user passes milestones selected by the architect. Timestamps are used to verify whether the data is synchronized or not. The first milestone would typically be the login screen. At login, the browser would send the timestamps of each entity stored in the browser's database. The application server processes the login and returns the response to the client. The application server then concurrently analyzes the timestamps and adds data to the client's update queue in order to keep it up to date. The data is

sent to the client on the next page response that is sent to the client. Each response to the client includes data updates which are in the form of JSON data. When updating data, the system follows the procedure shown in Figure 5.3. When updates are received, it is updated immediately in the NoSQL database and a response is sent to the client. In a concurrent operation, the update is sent to the RDBMS and the timestamp is updated at the NoSQL database. Once successful, the isDirty flag is set, and any new data which is in the update queue is sent to the client.

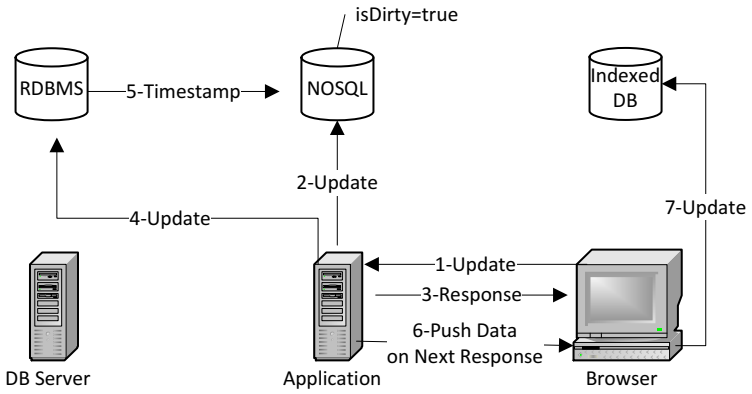

Figure 5.3: Passively consistent update.

5.2.2 Gradually Consistent

Gradual consistent entities have the property of being synchronized with the RDBMS in timed intervals. This increases the performance through batch processing which favors performance over consistency. A seen in Figure 5.4, passively consistent data items are queried from the NoSQL database. The application benefits from being queried from a fast dedicated database server. The update of gradually con-

sistent objects is considered infrequent, but it is frequently queried. The update process of a gradually consistent item can be seen in Figure 5.5. When the user chooses to update an item, the new data is immediately updated in the NoSQL database. This allows the application to return control to the user quickly while queuing transactions to be updated at the RDBMS. When the transaction is put in the transaction queue, a copy of the before and after objects are kept.

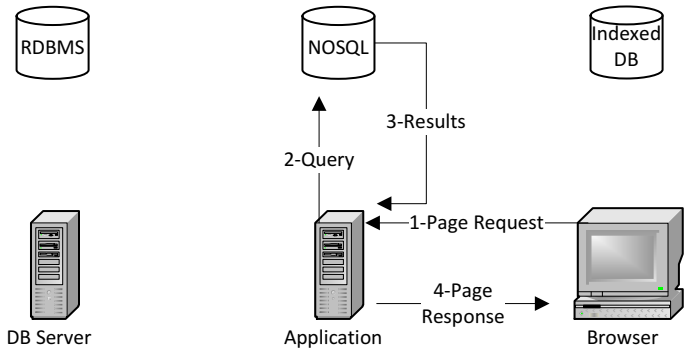

Figure 5.4: Gradually consistent query.

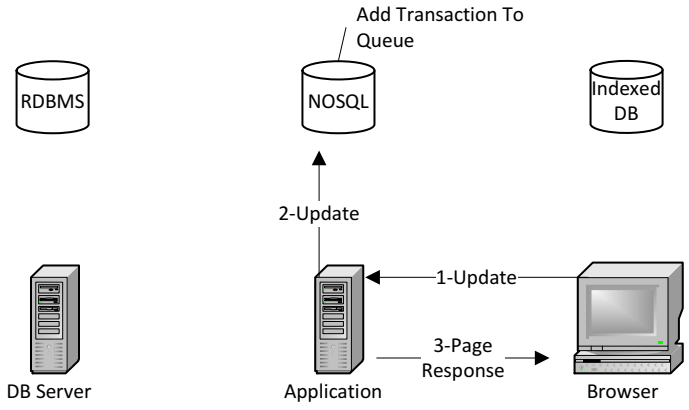

Figure 5.5: Gradually consistent update.

Up until now, the RDBMS is out of date with the updates that the user have executed. Since the gradually consistent data accepts inconsistent data for a certain amount of time while the system is collecting updates. Therefore the updating the RDBMS is triggered on an elapsed timer event. Once the timer has elapsed the system concurrently executes a batch update to the RDBMS as seen in Figure 5.6. Because data objects are not verified at the RDBMS on every update, a synchronization algorithm in will be employed to ensure the best possible consistency.

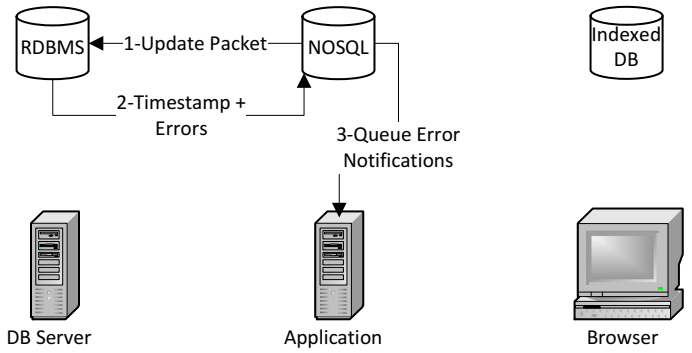

Figure 5.6: Gradually consistent timer elapsed.

5.2.3 Eventually Consistent

Eventually Consistent Entities are stored at the dedicated NoSQL database for fast data retrieval as seen in Figure 5.7. When the user updates the data, the process is executed as shown in Figure 5.8. The NoSQL database is updated and the response is given to the user, in a concurrent process, an update is initiated on the RDBMS. Eventual consistency uses timestamp comparisons to determine if the update should be accepted, rejected or merged. If the timestamp of the RDBMS ob-

ject and the NoSQL object is the same, the update is accepted as no changes were made to the data before If the timestamps differ, then an update has occurred on the RDBMS between data retrieval and update. In this case each field of the proposed update, and the RDBMS data which matches the timestamp. Also, a each field of the current RDBMS data is compared to the data matching the timestamp. If the changed fields intersect then, the data cannot be updated. If not, the data is merged by only changing the fields which have been updated in the current update operation. Ensuring not to overwrite any fields that have been changed previously.

When updates are processed on eventually consistent objects on the RDBMS, a broadcast message is sent to all registered NoSQL databases with the new data. This ensures fast updates of the NoSQL database without polling. Once the data is received it uses timestamp ordering to update data in the NoSQL database. This keeps the NoSQL database very close to actual consistency.

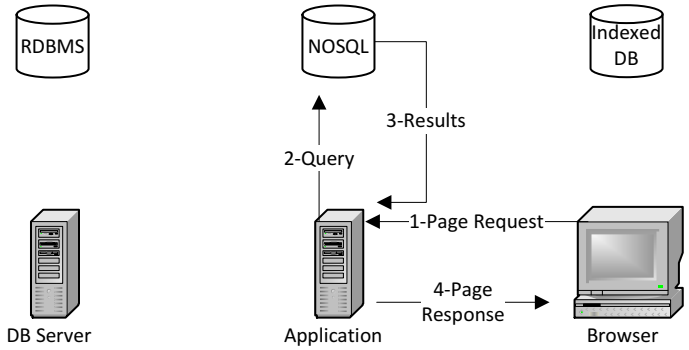

Figure 5.7: Eventually consistent query.

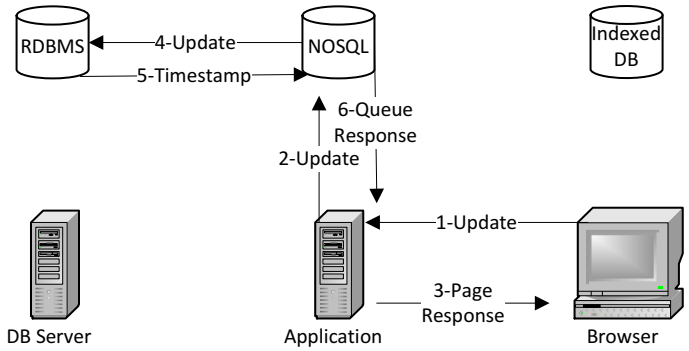

Figure 5.8: Eventually consistent update.

5.2.4 Always Consistent

Always consistent data is always queried and updated via the RDBMS server. It's operation is exactly the same as standard 3-tier architecture. The query process may be seen in Figure 5.9 and the update process in Figure 5.10. In this query the data which is in the application is always the data which is stored in the RDBMS. Concurrency control is handled completely by the RDBMS. Therefore, the user is subject to the delays and latency issues dictated by the current state of the RDBMS server.

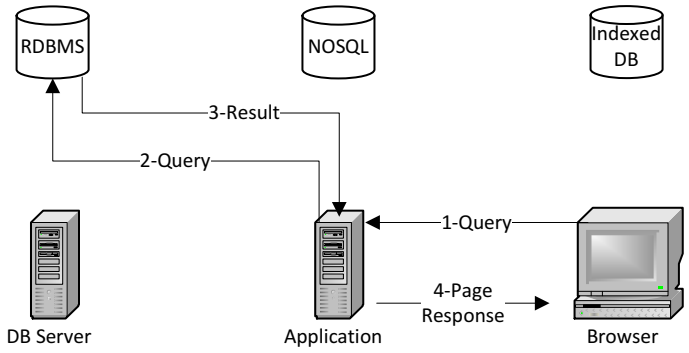

Figure 5.9: Always consistent query.

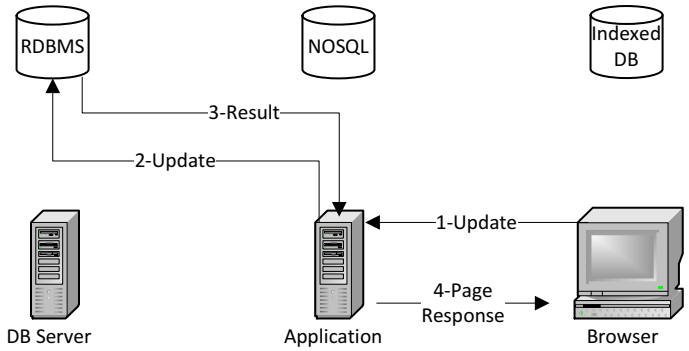

Figure 5.10: Always consistent update.

5.2.5 Immediately Consistent

Immediately consistent is the highest consistency level and offers consistency guarantees higher than typical web applications. When a user queries an immediate consistent data object, a lease request is registered for this item and the item is retrieved from the RDBMS and stored in memory. If any updates occur while a user is viewing it, the update is immediately sent to from the RDBMS to the application server. At which time, it is updated at the user's screen immediately. The browser uses long polling techniques to keep a connection to the application server open. This allows it to receive updates in realtime. Subsequent queries of this data by one or more users result in the copy of the item in memory being returned to the requester and that requester receives the same updates in real time as seen in Figure 5.11.

When a user wishes to modify an immediately consistent object, a lock must be acquired. The process to lock the item can be seen in Figure 5.12. Once this item is

locked, a lock ID is inserted into the lease entity. Any subsequent update attempt for this object must include the lock code. If the lock code is not included or is the wrong code, the update is rejected. If another user tries to lock the object while it is already locked, a null lock code is returned which keeps the user from locking this object. If the another user wishes to be put in queue to edit this object, they may choose to do so and the lock will be forwarded to the next user in order. This ensures only one user may update this record at a time.

Once the lock is acquired, the user may update the object as seen in Figure 5.13. Any update will be forwarded to all users who are viewing this object through the data leasing system and realtime system. Once the update is complete, the lock is automatically removed and may be requested by another user if any users are in the lock queue, a new lock code will be sent to the first user in the queue.

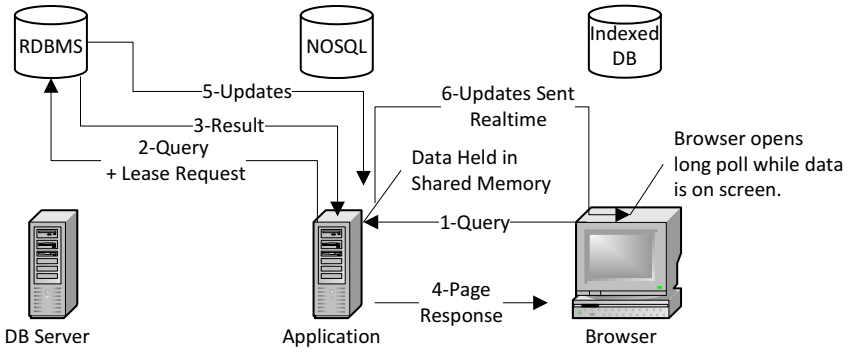

Figure 5.11: Immediate consistent query.

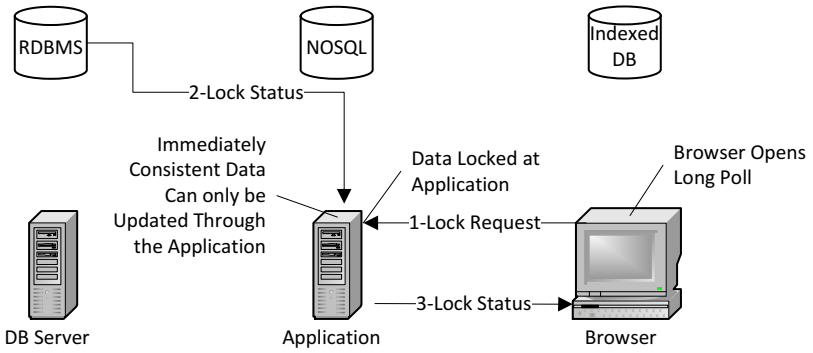

Figure 5.12: Immediate consistent lock.

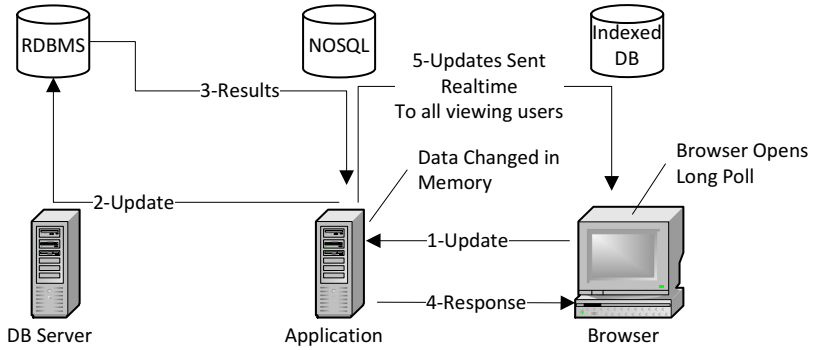

Figure 5.13: Immediate consistent update.

5.3 Integration in Business Application

To prove the value in a real world scenario a realistic business application has been built and tested with ADDA. The ADDA system was implemented in three tier architecture as seen in Figure 5.1.

For the RDBMS system, Microsoft SQL Server was used in an Athlon x64 system with 3GB RAM running the Windows Server 2008 operating system. The application tier was implemented on an Intel i5 system with 4GB RAM using Oracle Glassfish 4.0 application server and Mongo DB 2.4.4 running on the CentOS 6 operating system. The client is running on a Intel i3 system with 8GB RAM using Google Chrome 29.0 web browser running on the Windows 7 operating system. The data model used with the test application may be seen in Figure 5.14. This data model is based on the TPC Benchmark C (TPC-C) which simulates a real-world business application database. The TPC-C benchmark tests were used to simulate the complex activity of an OLTP activities on the RDBMS system. In addition, a separate TPC-C database was used to measure the performance of the running application. This accurately simulates access to an isolated database on a shared RDBMS server.

The system was implemented using Java Enterprise Architecture 7.0 (JEE), Java Server Faces 2.1 (JSF), Primefaces 3.4, Eclipselink Java Persistence Architecture 2.5 (JPA) and JQuery 1.8.1. JPA is a Object Relational Mapping (ORM) framework which seamlessly maps database objects to Java objects. In order to implement both the SQL and NoSQL database, a composite persistence unit was used which includes one persistence unit with all of the SQL Server entities and another per-

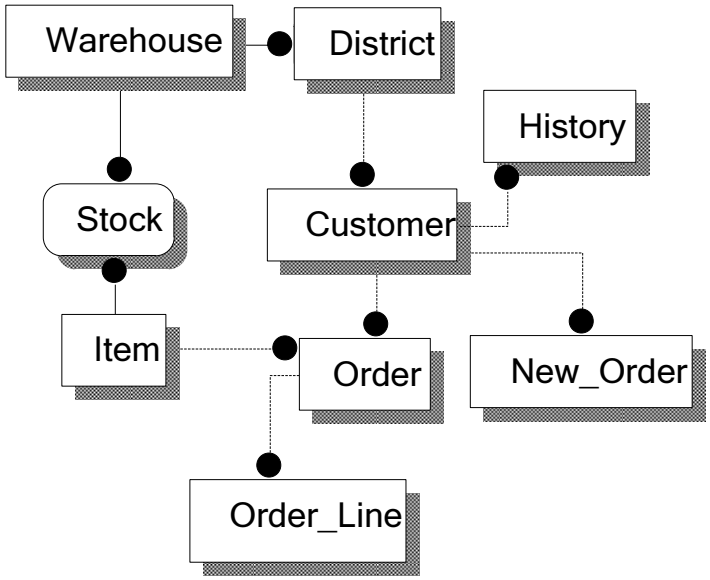

Figure 5.14: Test system data model.

sistence unit which includes the PC, GC and EC entities. Both persistence units extend an abstract class so an object from any consistency level can be used in the application with a high degree of location transparency.

Each entity of the test database was assigned a consistency level. The data model of the test database along with the assigned consistency levels, illustrated as two letter abbreviations, may be seen in Figure 5.15. This model allows for testing of the various parts of the system in a realistic OLTP environment.

In order to implement the system, each data tier must have the storage metadata shown in Figure 5.16. These items allow the ADDA system synchronize, instantiate and copy objects between database locations. During application startup, the ADDA system synchronizes PC, GC and EC entities by using storage metadata from the RDBMS and the NoSQL servers. The resulting inconsistencies are then updated to the NoSQL server to bring both databases to a consistent state.

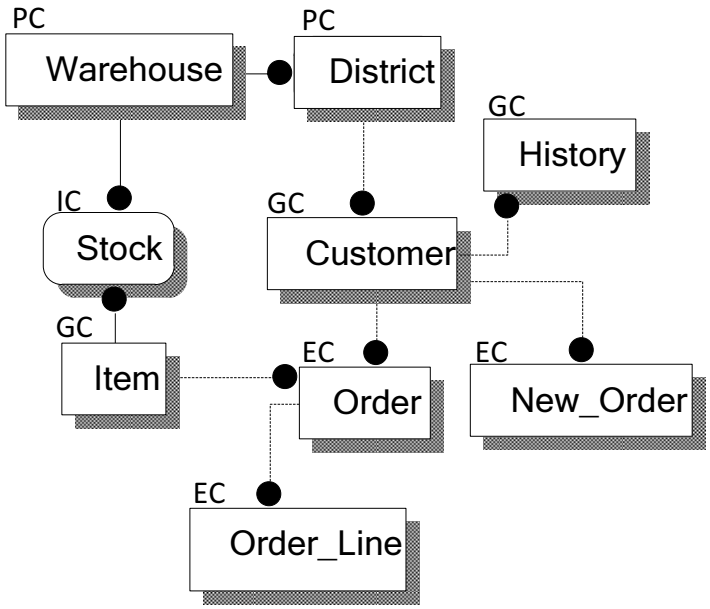

Figure 5.15: Test system data model with consistency levels.

The NoSQL server contains the PC, GC and EC entities in the same form as the RDBMS except the table names are prepended with the letter ”N”. This allows for easy differentiation when developing the application because it prefers convention over configuration.

All entities have a procedure which updates the updateTs in the StorageInfo entity as a part of their update process. In the RDBMS, this takes the form of an update trigger. The browser database uses a Javascipt component which updates the the browsers StorageInfo collection. Since the NoSQL database does not support triggers, the StorageInfo data is updated through a Java procedure.

The RDBMS procedure has additional functionality besided updating the timestamp which may be seen in Figure 5.17. This procedure checks if the updated object has a lock data lease against the object indicated as a leaseType ’LCK’. If the

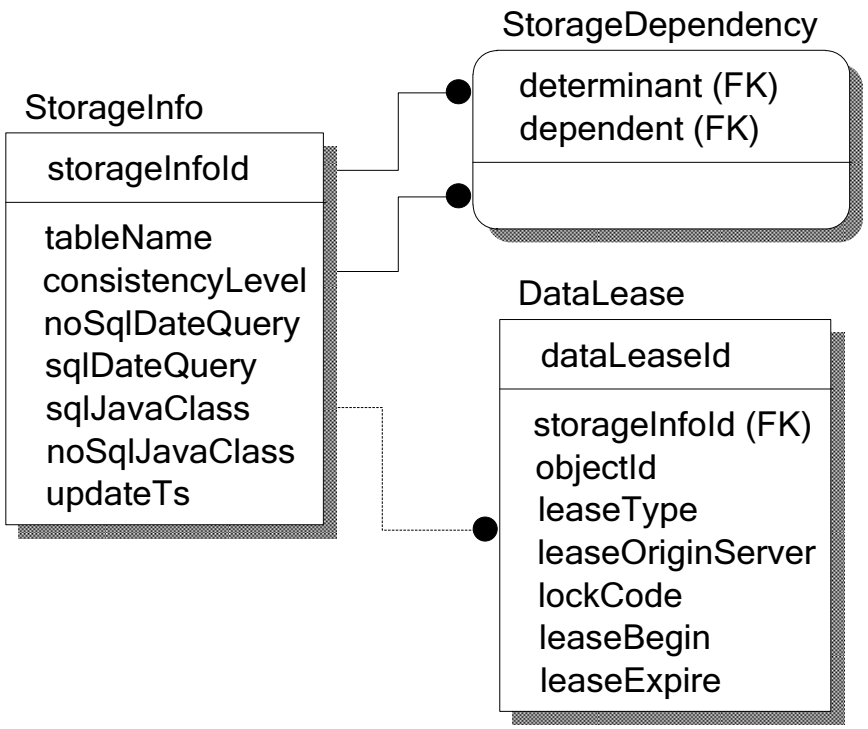

Figure 5.16: Storage meta-data included in each database location.

item is locked an error is thrown and the changes rolled back. If not, the procedure updates the StorageInfo updateTs attribute. Then, the procedure tests if the current object is an EC entity or an IC entity with a read data lease as indicated by the leaseType 'RD'. If either condition is true, a message with this entity and the new update timestamp is sent to the Data Leasing system via a message queue. This prompts the Data Leasing system to begin to update the NoSql database and synchronize it with the RDBMS server. Once that is complete, the procedure checks for any update dependencies through the StorageDependency entity. This allows for views of joined data to be externalized. If dependencies are found, the information is sent to the Data Leasing system to prompt it to synchronize the NoSql database.

The synchronization of the browser database takes place in both the client tier and the application tier. When a client logs in with their browser, the browser sends the StorageInfo data to the application server which prepares an update packet on a separate thread of execution. When the update is ready it is put on the user's update queue. This is then sent to the browser in the form of a JSON object transmitted in a hidden field. An example of this JSON object may be seen in Figure 5.18. In this example keys 1 and 2 are inserted or updated in table1 with the name and description attributes. When the Javascript component detects data in this field, it converts the JSON text into a Javascript object which contains a tables object that contains an array of objects with the table name, key and values. This data is iterated through and updated in the browsers database.

PC data is stored at both the NoSQL database and the Browser Database. The browser database is very limited in functionality and does not offer the complex

Algorithm 1: Update Procedure on RDBMS

1 TS = CURRENT_TIMESTAMP;
2 $SID \leftarrow \Pi_{(storageInfoId)}\sigma_{tableName=thisTableName}(StorageInfo)$;
3 **if** $thisRowId \neg \in$
$\Pi_{(rowId)}\sigma_{(leaseType='LCK',storageInfoId=SID,lockCode\neq thisLockCode)}(DataLease)$;
4 **then**
5 Throw ERROR 'THIS IMMEDIATELY CONSISTENT ITEM IS LOCKED.';
6 **end**
7 **else**
8 $CL \leftarrow \Pi_{(consistencyLevel)}\sigma_{storageInfoId=SID}(StorageInfo)$;
9 $StorageInfo \leftarrow \Pi_{(updateTs=TS)}\sigma_{storageInfoId=SID}(StorageInfo)$;
10 $LS \leftarrow \Pi_{(storageInfoId)}\sigma_{leaseType='RD',leaseBegin<TS,leaseExpire>TS,}(DataLease)$ **if** $CL =' IC'$ **then**
11 $DataLease \leftarrow$
$\Pi_{(lockCode\leftarrow thisLockCode,leaseType\leftarrow'LCK')}\sigma_{storageInfoId=SID,RowId=thisRowId}(DataLease)$;
12 **end**
13 **if** $CL =' EC' \vee (CL =' IC' \wedge SID \in LS)$ **then**
14 $UpdateMessagQueue \leftarrow (thisTableName, TS)$;
15 **end**
16 **foreach**
17 $CL \leftarrow \Pi_{(consistencyLevel)}\sigma_{determinant=SID}(StorageInfo \bowtie$
$StorageDependency_{(storageInfoId=dependent)})$;
18 $TN \leftarrow \Pi_{(tableName)}\sigma_{determinant=SID}(StorageInfo \bowtie$
$StorageDependency_{(storageInfoId=dependent)})$;
19 $DP \leftarrow \Pi_{(dependent)}\sigma_{determinant=SID}(StorageDependency)$;
20 **do**
21 **if** $CL =' EC' \vee (CL =' IC' \wedge DP \in LS)$ **then**
22 $UpdateMessagQueue \leftarrow (TN, TS)$;
23 $StorageInfo \leftarrow \Pi_{(updateTs=TS)}\sigma_{storageInfoId=DP}(StorageInfo)$;
24 **end**
25 **end**
26 **if** $CL =' IC$ **then**
27 $DataLease \leftarrow$
$\Pi_{(lockCode\leftarrow\emptyset,leaseType=\emptyset)}\sigma_{storageInfoId=SID,RowId=thisRowId}(DataLease)$;
28 **end**
29 **end**

Figure 5.17: Update Procedure on RDBMS

```
{ "tables" :
  [
    {"tablename" : "table1", "key" : "1", "value" :
      {"name" : "object1" , "description" : "description1"}
    },
    {"tablename" : "table1", "key" : "2", "value" :
      {"name" : "object2" , "description" : "description2"}
    }
  ]
}
```

Figure 5.18: Example JSON data to update the browser database.

query model enjoyed by RDBMS systems Nonetheless, it can be a very effective method to make data operations very responsive because queries do not have to be sent to the application server for processing. However, its limitations make it more useful for reference data in which complex joins are not needed. One use for this is replacing numeric keys with human readable fields. For example, the district id of 2 does not have a meaning to the user. Replacing the id with the district name requires a join at the database server which may be expensive. In order to offload this burden from the database server and reduce joins, the id number is sent to the browser in the html page along with the name of the entity and the display field. The browser looks up these items and replaces the number with the human readable name. Additionally, picklists may be displayed for the users input without calling the server resulting in responsive operation.

Since PC and GC level entities are subject to concurrent updates a LWW methodology is used to resolve such conflicts. Due to the fact that ORM frameworks may update an entire object at once, there is a danger of loosing updated attributes. This happens when updates are performed on two different attributes of the same

parent object. Although the two attributes are not in conflict, but their parent object are subject to WW conflicts. To resolve this, the merging algorithm shown in Figure C.3 has been employed. This algorithm checks if the object was modified between retrieve and update. If an update did occur, it compares the before and after snapshot of the proposed update, determines which fields have been modified and only changes those fields in the target object.

Algorithm 2: Synchronization Algorithm

Data: *UpdateTransactionList* = List of update transactions collected by the application including before and after versions of the data objects.

```
1  foreach Transaction in UpdateTransactionList do
2      RDBMSRecord = Select current record from RDBMS;
3      if RDBMSRecord.UpdateTimestamp == Transaction.OriginalTimestamp
       then
4          RDBMSUpdate( Transaction.after.allFields);
5          else
6              for i ← to Transaction.FieldCount do
7                  if Transaction.before.field[i]!=Transaction.after.field[i] then
8                      UpdateList.add( Transaction.after.field[i]) ;
9                  end
10                 RDBMSUpdate(UpdateList);
11             end
12         end
13     end
14 end
```

Figure 5.19: Synchronization Algorithm

GC, PC and EC entities are stored in the NoSQL database. NoSQL databases do not offer the full functionality of a RDBMS database. For example, MongoDb is a BSON document datastore which means that data is stored and processed as BSON documents, not relations. Therefore, joins are not natively supported.

Unlike RDBMS tuples, a BSON document may include objects and arrays. This means that NoSQL databases violate Codd's 1st Normal Form (1NF) [14] because an attribute can contain more than one single value.

In NoSQL databases, an attribute can contain an array or an object or even an array of objects. This is useful for high performance as expensive joins are avoided and related data may be stored in a single document. While not adhering to 1NF increases flexibility and speed, it opens up the possibility of update anomalies. Therefore, to update related data, one would have to find all objects which include the subject data and update them as well. On the other side, data in Cod's 3rd Normal Form, the standard for relational data, suffers from difficulties in representing data accurate to transactional time as referenced in [31]. NoSQL databases do not suffer from that because updated reference data is only updated at a point of time forward unless older objects are explicitly modified. NoSQL databases store the reference data as a part of the parent data. Storing parent and reference data in the same document in the ADDA system will cause inconsistencies between the NoSQL data and the RDBMS data.

To combat inconsistencies due to disparate storage methodologies between the RDBMS and NoSQL, data is joined at the RDBMS server as a view. This view is then externalized in the NoSQL database resulting in externalized views which is consistent with RDBMS data. Equation 5.1 illustrates the three joined entities which may represent a view in our application. Let R, S and T be relations. A change in any of the attributes of R, S and T will result in a change in the result of the R joined to S joined to T. For this reason, the concept of entity dependence was included in the storage metadata as seen in Figure 5.16. This allows the application

to correctly update an externalized view based on a change in the underlying table data.

$$\begin{aligned} R \bowtie S_{(A=A')} \bowtie T_{(B=B')} &\rightarrow R \times S \times T\sigma_{(A=A',B=B')} \\ \therefore \Delta R \vee \Delta S \vee \Delta T &\rightarrow \Delta(R \bowtie S_{(A=A')} \bowtie T_{(B=B')}) \end{aligned} \tag{5.1}$$

The test system was constructed and tests were evaluated to determine the effects of the ADDA architecture on the application and the RDBMS.

5.4 Performance Experiments

To measure the difference in responsiveness to the user, a Javascript component was built to record the exact time when a user requests an object and the time when the entire response is finished loading including all Javascript functions executed and the result is displayed to the user. Unlike server-side measurements, this method measures the actual response time that the user experiences.

The response-time test used a 3 table join data grid with a total table size of 60,000 records as its test data. Five test queries were executed for each scenario and their response time was recorded. Three scenarios were compared.

1. RDBMS query using ORM relationships. This method used the ORM to join data and return the results.

2. RDBMS query using views. This method defined a view which incorporated the three table join and the ORM system treats the view as a separate entity.

3. NoSQL query using externalized views.

These three scenarios demonstrate the performance difference between the standard ORM method, mapping an ORM object to a SQL view and using the NoSQL to retrieve the data. The NoSQL query test tests GC and EC query performance. Each of these tests were conducted under various load conditions using the TPC-C benchmark to simulate user traffic on the RDBMS under controlled conditions. Figure 5.20 shows the result of this test. Throughout the test, the response time of the RDBMS was higher than the NoSQL queries. As expected the NoSQL was unaffected by the traffic on the RDBMS server and consistently resulted in response times of around 300ms even though RDBMS load traffic originated from the same server as the NoSQL database. Using views is a vast improvement over standard ORM relationships; however, the data shows the most responsive performance comes from the NoSQL database using GC or EC entities. This demonstrates the advantages of this design.

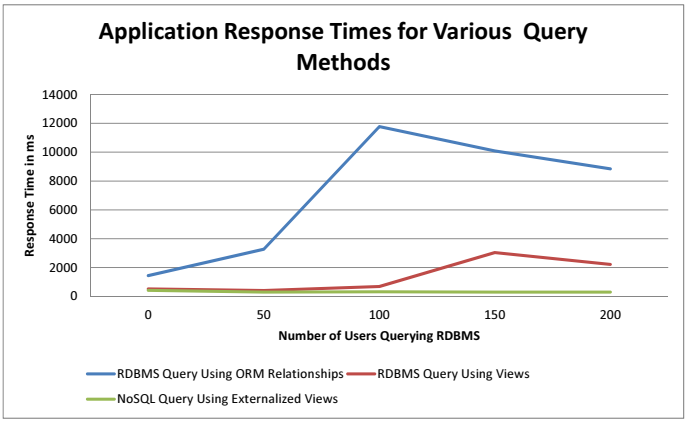

Figure 5.20: Application response time for RDBMS vs ADDA architecture.

The synchronization of records between the RDBMS and the NoSQL database

was tested. In this test the measurement of time began at the first step of the synchronization which is the retrieval of the StorageInfo objects from both databases. The two StorageInfo objects are then compared, the out of sync items are identified, retrieved from the RDBMS and updated to the NoSQL server. This procedure is used for PC,GC and EC entities. The results of this test can be seen in Figure 5.21. For 20 records, the update time was 20ms/record. When a larger entity with 30,000 records were updated, the updated time per record decreased to 2ms/record. When 60,000 records were updated, the update time per record was reduced to 1.8ms/record. This represents the cost of copying data from the RDBMS to the NoSQL server. This cost is only applicable during startup or during other milestones or timer events of PC and GC entities respectively.

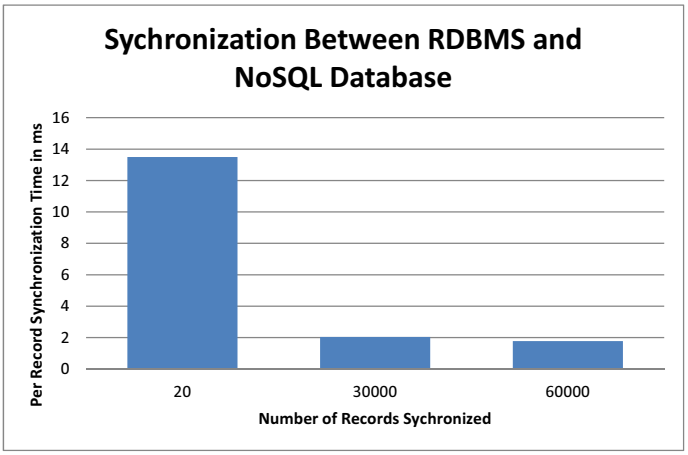

Figure 5.21: Synchronization time between RDBMS and NoSql databases.

Both EC and IC entities use a message queue to update objects in real time. The difference between EC and IC is that EC objects are stored at the NoSQL database and IC objects are stored at the RDBMS database. When a user is viewing an IC

object, they acquire a read lease on the object. When the user has a read lease, any update to the object is sent to the user's browser in real time. In the case of IC objects, the data is sent from the RDBMS database and loaded directly into memory. Unlike IC objects, the EC entities always get updated in real time. The EC also objects have an additional step to update the NoSQL database. Therefore, the performance of the Data Leasing system real time update mechanism was tested. The test measured the time of update on the RDBMS and compared it to the time in which the IC and EC object updates were completed. The results of this test is recorded in Figure 5.22. The response time for these update is very low below 100 users where the database response time peaks. The RDBMS response time lowers somewhat when 150-200 users are actively using the RDBMS. According to this results, the IC and EC updates take a maximum of two seconds to complete their operation.

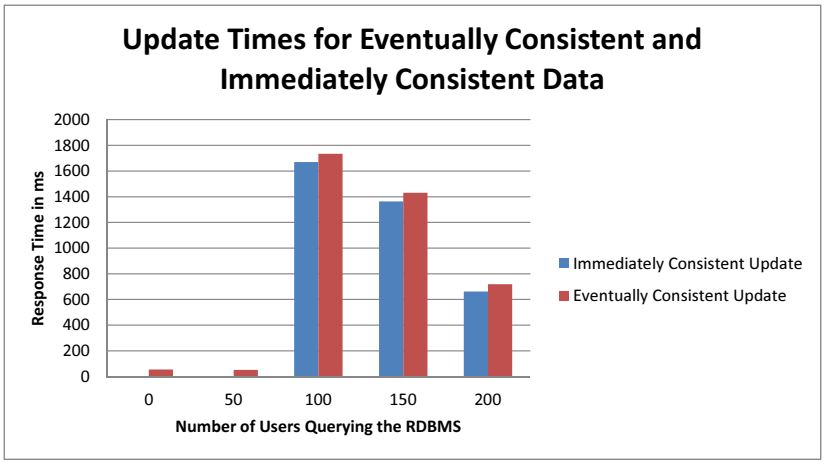

Figure 5.22: Time for EC and IC immediate update.

The PC entities leverage the in-browser database to improve response time. The

Javascript component mentioned above was used to measure the response times for various scenarios. The test was to query a list of data which was required to be joined to a reference table. Two scenarios were tested. One scenario used the standard ORM joins to retrieve the data from the RDBMS. The second scenario used the same data from the RDBMS without the join to the reference data. In this scenario, the reference data is a PC entity stored at the browser. The results can be seen in Figure 5.23. As the load on the RDBMS server was increased the advantages of using the browser-based PC data is clear. At 200 active users, the PC data response time was 3x less than using the RDBMS ORM joins. The cost to load the data into the browser database has also been tested. Figure 5.24 shows that an average update time for 20 records is approximately 2ms per record. This concludes that the benefit far outweighs the cost of maintaining the data at the client.

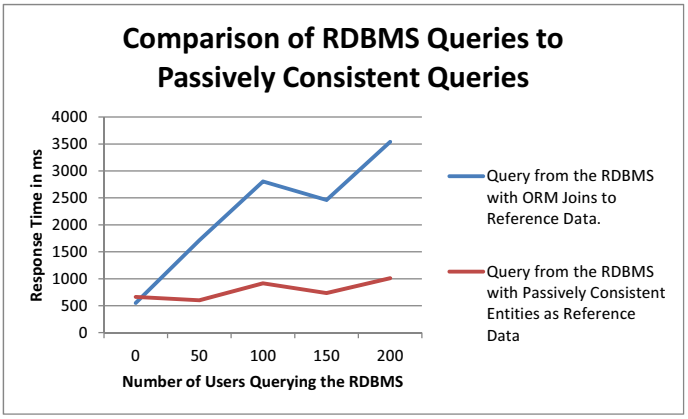

Figure 5.23: Application response time between RDBMS and browser database.

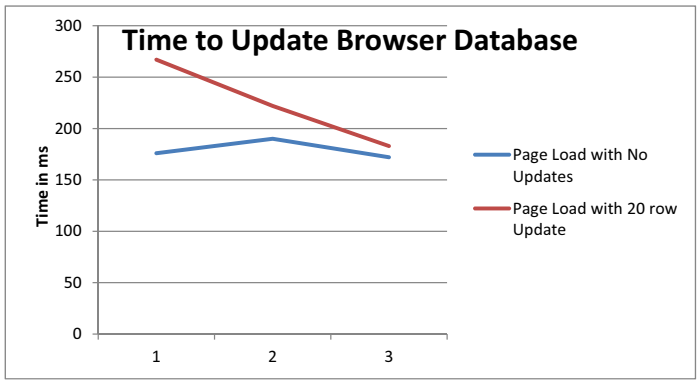

Figure 5.24: Update time for browser database.

Chapter 6

Conclusions

The experiments in this paper have demonstrated the advantages of using the ADDA architecture. The ADDA architecture allows the architect to design a single data model and distribute the data based on the acceptable consistency level. This architecture can improve the responsiveness of an application 10 times or more than using a single shared RDBMS server alone.

The Passively Consistent level allows one to leverage the browser-based database which is part of the HTML5 specification to achieve a richly responsive application. Query and display times for data using this method are on average three times less than using an ORM system to query the reference data from the RDBMS system. The cost to maintain the data in the browser-based database is low averaging two milliseconds per record. Because of the limitations of the browser-based database, Passively Consistent data should be limited to small sized reference data which does not need the join functionality.

Larger reference data should be stored as Gradually Consistent data. This al-

lows for increased performance from the NoSql database which can return results up to 10 times faster than a heavily loaded RDBMS. Gradually Consistent entities are updated based on periodic intervals. The cost to updated the data in the NoSql database in terms for time can be as low as two milliseconds per record. The cost to update these items is very low as compared to the benefits. These data entities work in batch update mode; therefore, they employ optimistic concurrency in the form of Last Writer Wins methodology. This is enhanced by a merging algorithm; however it is advisable to use this consistency level on data that does not change often to maximize its high gain, low cost profile.

Eventually Consistent data can reap even more benefits in terms of response time. The data has shown that Eventually Consistent data can be 30 times faster than joining relational data on a loaded RDBMS server. Utilizing the concept of externalized views, joins are processed only one time, then externalized on the NoSql database. Because Eventually Consistent Data is updated immediately via inter process communication, the cost is higher to maintain the data in the NoSql database. However, since a data refresh may take a maximum of two seconds on a heavily loaded RDBMS server, the Eventually Consistent data should always be at an acceptable consistency level for data which is not time sensitive at the second or millisecond level. The concept of table dependencies make externalized views on a NoSql server possible.

Immediately Consistent data is held on a read lease whenever a user opens an Immediately Consistent record. This allows for real time updates without re-querying the RDBMS server. The first query for this consistency level comes directly from the RDBMS server. Subsequent queries are retrieved directly from

memory. With externalized locking Immediately Consistent objects ensure blockless exclusivity reducing the possibility of deadlocks. The data has shown that the maximum update time for Immediately Consistent data is approximately 1.6 seconds on a heavily loaded RDBMS.

The ADDA architecture is a basis for continuing research on methodology for design of distributed database applications.It favors small, fast databases which bring the data closer to the client while maintaining acceptable levels of consistency can be the key to achieving responsive applications and lowering the load on RDBMS servers. Further research into more complex models and dependencies will help us utilize this new architecture in a myriad of applications.

This paper is part of my on-going research in using middle-ware technologies to integrate heterogeneous data storage systems. I first published on this concept a part of the 2012 SAIS conference [23]. Versions of this paper are currently under review for publishing in the Communications of the ACM with plans to publish in other venues as well.

Bibliography

[1] *BSON Specification 1.0*. http://bsonspec.org/#/specification [Accessed 11 August 2013].

[2] *Distributed Transaction Processing: The XA Specification*. http://pubs.opengroup.org/onlinepubs/009680699/toc.pdf [Accessed 01 September 2013].

[3] *SQL Server 2008 R2 Books Online*. Microsoft Corporation. http://www.microsoft.com/en-us/download/details.aspx?id=9071 [Accessed 10 August 2013].

[4] *The MongoDB 2.4 Manual*. http://docs.mongodb.org/master/MongoDB-manual.pdf [11 August 2013].

[5] *Windows PC Accelerators*, Oct. 2010. http://msdn.microsoft.com/en-us/library/windows/hardware/gg463388.aspx [Accessed 5 July 2013].

[6] *About V8*, 2012. http://developers.google.com/v8/intro [Accessed 16 August 2013].

[7] *HTML 5.1 Specicification*, 08 2013. http://www.w3.org/html/wg/drafts/html/master [Accessed 16 August 2013].

[8] L. Ashdown and T. Kyte. *Oracle Database Concepts11g Release 2 (11.2)*. Oracle. http://docs.oracle.com/cd/E11882_01/server.112/e25789.pdf [Accessed 8 August 2013].

[9] A. Avram. Hybrid SQL-NoSQL Databases Are Gaining Ground. *www.infoq.com*, Feb. 2012. http://www.infoq.com/news/2012/02/Hybrid-SQL-NoSQL [Accessed 10 September 2013].

[10] J. M. Barnes. Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications. Master's thesis, Macalester College, Apr. 2007. http://digitalcommons.macalester.edu/mathcs_honors/6/ [Accessed 4 August 2013].

[11] R. Billinton and R. N. Allan. *Reliability evaluation of engineering systems*. Plenum press New York, 1983.

[12] E. A. Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[13] R. Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[14] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, Jan. 1983.

[15] T. M. Connolly. *Database systems: a practical approach to design, implementation, and management*. Pearson Education, 2005.

[16] C. de la Torre, U. Zorrilla, M. A. Ramos, and J. Calvarro. *N-Layered Domain-Oriented Architecture Guide with .NET 4.0*. Microsoft Corporation, first edition.

[17] P. J. Deitel and H. M. Deitel. *Internet and World Wide Web: How to program*. Prentice Hall, New Jersey, U.S.A., 2009.

[18] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, Nov. 1976.

[19] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[20] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 1 edition, Nov. 2002.

[21] C. Gerlhof and A. Kemper. A multi-threaded architecture for prefetching in object bases. *Advances in Database TechnologyEDBT'94*, pages 351–364, 1994.

[22] J. Gray. The Transaction Concept: Virtues and Limitations. 1981. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.5051 [Accessed 3 August 2013].

[23] M. Grecol. STANDARD DATA MODEL FOR CUSTOMS EDI FILINGS. Southern Association of Information Systems, 2012.

[24] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: a main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, Nov. 2010.

[25] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[26] W.-S. Han, K.-Y. Whang, and Y.-S. Moon. A formal framework for prefetching based on the type-level access pattern in object-relational DBMSs. *Knowledge and Data Engineering, IEEE Transactions on*, 17(10):1436–1448, 2005.

[27] Z. He and A. Marquez. Path and cache conscious prefetching (PCCP). *The VLDB Journal*, 16(2):235–249, Apr. 2007.

[28] E. Houts. Starnine combination is capable, complex solution. *InfoWorld*, 19(44):83–86, 1997.

[29] S. Khemmarat, R. Zhou, L. Gao, and M. Zink. Watching user generated videos with prefetching. In *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, pages 187–198, New York, NY, USA, 2011. ACM.

[30] W. Kim, B.-J. Choi, E.-K. Hong, S.-K. Kim, and D. Lee. A taxonomy of dirty data. *Data Mining and Knowledge Discovery*, 7(1):81–99, 2003.

[31] C. Knowles. 6NF Conceptual Models and Data Warehousing 2.0. Southern Association of Information Systems, 2012.

[32] J. Lee, H. Kim, and R. Vuduc. When Prefetching Works, When It Doesn 't, and Why. *ACM Trans. Archit. Code Optim.*, 9(1), Mar. 2012.

[33] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Enhancing Edge Computing with Database Replication. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 45–54, 2007.

[34] C. Mohan. Tutorial: Advanced transaction models-survey and critique. In *ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, 1994.

[35] C. Nance, T. Losser, R. Iype, and G. Harmon. Nosql vs rdbms-why there is room for both. 2013.

[36] D. Pritchett. BASE: An Acid Alternative. *Queue*, 6(3):48–55, May 2008.

[37] S. K. Rahimi and F. S. Haug. *Distributed Database Management Systems: A Practical Approach*. John Wiley & Sons, Inc, 2010.

[38] P. Rösch, L. Dannecker, F. Färber, and G. Hackenbroich. A storage advisor for hybrid-store databases. *Proc. VLDB Endow.*, 5(12):1748–1758, Aug. 2012.

[39] C. Strauch, U.-L. S. Sites, and W. Kriha. NoSQL databases. *URL: http://www.christof-strauch. de/nosqldbs. pdf (07.11. 2012)*, 2011.

[40] R. H. Thomas. A Majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

Appendix A

Browser Database Components

```
public String updateBrowser (HashMap<String,Date> status) throws
ClassNotFoundException, UnknownHostException{
    Query q = getSqlEm().createNamedQuery("NStorageInfo.findall",
            NStorageInfo.class);
    noSqlStorageInfo = q.getResultList();
    String output = "{ \"tables\" : [";
    for (NStorageInfo ni : noSqlStorageInfo) {
        if(ni.getConsistencyLevel().equalsIgnoreCase("PC")){
            Date updateLevel = status.get(ni.getTableName().toUpperCase());
            if (updateLevel==null)updateLevel = new Date(0);
            System.out.println("Updating " + ni.getTableName() +" " +
             ni.getUpdateTs() +
ni.getUpdateTs());
            MongoClient mongoClient = new MongoClient(
            "192.168.254.11",27017 );
            DB db = mongoClient.getDB("tpccnosql");
            System.out.println(db);
            DBCollection coll = db.getCollection("N" + ni.getTableName().
toUpperCase());
            System.out.println(coll);
            BasicDBObject query = new BasicDBObject("updateTs",new
            BasicDBObject ("$gt",updateLevel ));
            System.out.println(query);
            DBCursor cursor = coll.find(query);
            System.out.println(cursor);
            boolean first = true;
            try {
                   while(cursor.hasNext()) {
                       DBObject d = cursor.next();
                       if (!first) output += ", ";
                       output += "{ \"tablename\" : \"" + ni.getTableName().
   toLowerCase() + "\" , \"key\" : \"" + d.get("_id") +
                              "\", \"value\" : " + d.toString() + "}";
                       first = false;
                       System.out.println(output);
                   }
               } finally {
                    output += "]}";
                   cursor.close();
               }
        }

    }
    return output;
}
```

Figure A.1: Procedure to create browser-based database update message at application server.

```
$().ready(function(){
var dbOpenPromise = $.indexedDB("orderSystem");
  dbOpenPromise.done(function(db,event){
    updatedbObj();
    lookupValues();
    pageLoaded();
});
dbOpenPromise.fail(function(error, event){
    alert('ERROR' + error.message);
    alert('ERROR Evt' + event);
});
});
```

Figure A.2: Procedure to open browser-based database.

```
function updatedbObj(){
  updateObject = $.parseJSON(unescape($("[id*=updatedb]").val()));
  if(updateObject) $.each(updateObject.tables,function(index,value){
     $.indexedDB("orderSystem").objectStore(
     value.tablename,true).put(value.value, value.key);});
}
```

Figure A.3: Procedure to update browser-based database.

```
function sendMetaData(){
  var Jsonvar;
  $.indexedDB("orderSystem").objectStore("StorageInfo").get().then(
    function(item){
      Jsonvar+=JSON.stringify(item);
      });
  $("[id*=storageInfo]").val(Jsonvar);
}
```

Figure A.4: Procedure to send browser-based database metadata to application server.

```
function lookupValues(){
  $("[id*=pclookup]").each(
    function(index){
    var s=$(this).text().split(":");
    var o = this;
    $.indexedDB("orderSystem").objectStore(s[0]).get(s[1]).then(
      function(item){
        $(o).text(item[s[2]]);
      });
    });
}
```

Figure A.5: Procedure to lookup a tuple from browser-based database and replace the text with the query results.

```
<p:dataTable id = "HistoryData" var="o"
value="#{orderController.sHistory}"
paginator="true" rows="10"
               paginatorTemplate="{CurrentPageReport}  {FirstPageLink}
               {RowsPerPageDropdown}"
               rowsPerPageTemplate="5,10,15">
      <f:facet name="header">
          Ajax Pagination
      </f:facet>
        <p:column headerText="o.hCId.cLast">
          <h:outputText value="#{o.ohCId.cLast}" />
      </p:column>
          <p:column headerText="o.hCId.cFirst">
          <h:outputText value="#{o.ohCId.cFirst}" />
      </p:column>
                <p:column headerText="o.hDId.dName">
          <h:outputLabel id = "pclookup"
          value="District:#{o.hDId}:dName" />
      </p:column>
                <p:column headerText="o.hDId.dCity">
           <h:outputLabel id = "pclookup2"
           value="District:#{o.hDId}:dCity" />
      </p:column>
                <p:column headerText="o.hData">
          <h:outputText value="#{o.hData}" />
      </p:column>
                <p:column headerText="o.hDate">
          <h:outputText value="#{o.hDate}" />
      </p:column>
 </p:dataTable>
```

Figure A.6: Example JSF code to present keys in document for browser-based database to lookup.

Appendix B

RDBMS - NoSQL Synchronization

```
public void SyncNoSql(String level) throws ClassNotFoundException {
        Query q = getSqlEm().createNamedQuery("StorageInfo.findall",
                StorageInfo.class);
        sqlStorageInfo = q.getResultList();
        q = getSqlEm().createNamedQuery("NStorageInfo.findall",
                NStorageInfo.class);
        noSqlStorageInfo = q.getResultList();

        for (StorageInfo si : sqlStorageInfo) {
            if (si.getConsistencyLevel().equals(level)) {
                NStorageInfo ni = null;
                for (NStorageInfo n : noSqlStorageInfo) {
                    if (n.getTableName().equals(si.getTableName())) {
                        ni = n;
                        break;
                    }
                }
                SyncOneEntity(si,ni);
        }
        }
DataListener listener = new DataListener();
listener.setDl(this);
Thread T = new Thread(listener);
T.start();
    }
```

Figure B.1: Procedure to synchronized RDBMS and NoSQL at startup.

```
public void SyncOneEntity(StorageInfo si, NStorageInfo ni) throws
ClassNotFoundException{
    em.clear();
    Query q;
    int counter = 0;
    long start = System.currentTimeMillis();
    if (ni != null & ni.getUpdateTs().before(si.getUpdateTs())) {
        System.out.println("Updating " + ni.getTableName() +
         si.getUpdateTs() + ni.getUpdateTs());
        q = getSqlEm().createNamedQuery(
                si.getSqlJavaClass().trim() + ".findUpdatedObjects",
                Class.forName(si.getSqlJavaClass()));
        q.setParameter("uTimeStamp", ni.getUpdateTs());
        List<Object> updateObjects = q.getResultList();
        getSqlEm().getTransaction().begin();
        for (Object fromObj : updateObjects) {
            try {
                Object toObj = Class
                        .forName(si.getNoSqlJavaClass())
                        .newInstance();
                Date uDate = copyProperties(fromObj, toObj);
                System.out.println(fromObj.toString() + ": "
                        + toObj.toString());
                if (uDate != null && uDate.after(ni.getUpdateTs())){
                    ni.setUpdateTs(uDate);
                    System.out.println("UpdatedDate" + uDate);
                }
                em.merge(toObj);
                em.merge(ni);
                if(counter%1000==0){
                    getSqlEm().getTransaction().commit();
                    getSqlEm().getTransaction().begin();
                }
                counter++;
            } catch (InstantiationException e) {e.printStackTrace();
            } catch (IllegalAccessException e) {e.printStackTrace();}
            System.out.println("Updated " + counter
                    + " Objects in "
                    + (System.currentTimeMillis() - start)
                    + "milliseconds");
        }
        em.merge(ni);
        em.getTransaction().commit();
        System.out.println("Updated " + counter + " Objects in "
                + (System.currentTimeMillis() - start)
                + "milliseconds");
        System.out.println("Time: " + System.currentTimeMillis());
    }
}
```

Figure B.2: Component to synchronize NoSQL using timestamps.

Appendix C

RDBMS Update Publisher and Listener

```
ALTER TRIGGER [dbo].[HISTORY_EM]
   ON  [dbo].[HISTORY]
  AFTER INSERT,DELETE,UPDATE
AS
BEGIN
SET NOCOUNT ON;
DECLARE @tableid varchar(35);
DECLARE @Message VARCHAR(128);
DECLARE @SBDialog uniqueidentifier;
DECLARE @dt as datetime;
set @dt = GETDATE();

update HISTORY set updateTs = @dt where h_c_id2 =
(select h_c_id2 from inserted)
BEGIN DIALOG CONVERSATION @SBDialog
FROM SERVICE SBSendService
TO SERVICE 'SBReceiveService'
ON CONTRACT SBContract
WITH ENCRYPTION = OFF
DECLARE tableids CURSOR
    FOR
select s2.tableName from StorageInfo si
inner join  StorageDependency sd on si.storageId = sd.determinant
inner join StorageInfo s2 on sd.dependent = s2.storageId
where si.tableName= 'HISTORY'
union
select tablename from StorageInfo where tableName = 'HISTORY';
OPEN tableids
FETCH NEXT FROM tableids INTO @tableid
WHILE @@FETCH_STATUS =0
BEGIN
update StorageInfo set updateTs = @dt
WHERE tableName = @tableid;
set @Message = (SELECT @tableid + ',' + cast(DATEDIFF(s,
'1970-01-01 00:00:00',
 @dt)as varchar(12)) + right(CONVERT(VARCHAR(26),@dt, 114),3));
SEND ON CONVERSATION @SBDialog MESSAGE TYPE SBMessage (@Message);
--select @Message;
FETCH NEXT FROM tableids INTO @tableid
END
CLOSE tableids;
DEALLOCATE tableids;
-- Send messages on Dialog
END
```

Figure C.1: Trigger to send update message to application during update.

```
private String getMessage(){
    String output="";
    DriverManager dm = null;
    Connection conn = null;
    try {
         conn = dm.getConnection("jdbc:sqlserver://192.168.254.10:1433;
         databaseName=tpcc2", "sa","anggita8264");
         System.out.println("Connected");
        Statement statement = conn.createStatement();
         System.out.println("Waiting for Queue");
        ResultSet rs = statement.executeQuery("waitfor ( RECEIVE TOP(1)
        CONVERT(VARCHAR(MAX), message_body) AS Message FROM
        SBReceiveQueue )");
System.out.println("MESSAGE \t " + System.currentTimeMillis());
        rs.next();
output  = rs.getString(1);
    } catch (SQLException e) {
        e.printStackTrace();
        try {
            if (conn!=null && !conn.isClosed()) conn.close();
        } catch (SQLException e1) {
            e1.printStackTrace();
        }
    }
    finally
    {
        if(conn!=null)
            try {
                conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
    }
    return output;
}
```

Figure C.2: Procedure to receive update message from RDBMS.

```
private void getUpdates(String message){
    String args[] = message.split(",");
    if(args.length>1){
        String name;
        long time;
        StorageInfo s=null;
        NStorageInfo n=null;
        name = args[0];
        time = Long.parseLong(args[1]);
        if (name.equals("")||time==0) return;
        for(StorageInfo si: dl.getSqlStorageInfo()){
            if(si.getTableName().equalsIgnoreCase(name)){
                s= si;
                break;
            }
        }
    for(NStorageInfo ni: dl.getNoSqlStorageInfo()){
            if(ni.getTableName().equalsIgnoreCase(name)){
                n= ni;
                break;
            }
    }
    if(n==null || s==null) {
        System.out.println("Cant' find ID " + name);
        return;
    }
    System.out.println(s.getUpdateTs());
    s= dl.getSqlEm().find(s.getClass(),s.getStorageId());
    System.out.println(s.getUpdateTs());
        System.out.println("Updating " + s.getTableName() +
        "\t" + args[1]);
        try {
            dl.SyncOneEntity(s, n);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

    }
}
```

Figure C.3: Procedure to synchronize NoSQL based on received update message.