On Datastore Support for Agile Software Development

Jie Liu
Computer Science Division, Western Oregon University, Monmouth, Oregon, 97361, U.S.A.

Keywords: Software Development, Agile Software Development Process, Databases, Database Management Systems, MongoDB, and NoSQL.

Abstract: Most database-base applications support two main sets of features: customer facing transactional capabilities such as purchasing books at an online bookstore and functionality required by managers and business analysts such as identifying trends in sales data by combing through aggregated sales data. The conventional approach of having just one main database to support both features greatly restrict developers freedom in applying the best approach to quickly implement new features, to enhance existing features, or to mend defects because any attempt in changing database schema means code and test cases modifications in many places and may even require a large amount of effort in testing. Such an inherited resistance in changing introduced by data store does not fit the evolutionary development nature of Agile software development methodology. We argue that we should consider having at least two databases: one support transactional capabilities and the other support reporting and possible data warehouse needs, and will show how such an approach supports the Agile software development methodology.

1 INTRODUCTION

With increasingly sophisticated IDEs, programming languages, knowledge, enough bad experiences in trying other software development methodologies by both the developers and management teams, and strong support of success stories favour Agile software development process, more and more software teams are adopting Agile software development process. In a nutshell, in our view, the benefits of Agile methodologies are to promote short development cycles, encourage communications and cooperation, and accommodate changes in requirements and scope. Each sprint is short and is focused on the most important issues at the time so the team can find out what work and what are the challenges quickly. Due to short sprint, changes on requirements are reflected promptly. Scrums or other frequent short meetings encourage communications and cooperation among the developers and also between developers and key stakeholders. The use of product backlog brings all shareholders' requirements together at one place so everyone can see. Finally, the end of sprint demo is an effective way to show the progress of the development team to the management team and other shareholders, not to mention it is also an effective vehicle for the product owners to interact directly with the developers and testers and to validate whether the implementation matches with what is expected. For projects that require a dozen or so co-located developers, Agile software development process has proven to be effective and lead to successful projects (Turk, 2002).

In just a few years, we have witnessed drastic change in the acceptances of Agile software development methodology. Right now, all software teams in our organization have adopted Agile software development methodology. The members of our management team are accustomed to attend our demos at the end of each sprint. Even the support and operation teams have daily scrums so both the team members and managers can have a good read on the vital signs about the team’s current state and team members' activities.

We understand and agree that Agile software development process has its limitations such as unfit to support large and distributed teams, unfit to support large and complex software application, and is claimed to be unfit for developing safety-critical software **(**Turk**,** 2002**)..** However, we are not here to discuss the limitations. Instead, we would like to discuss our experience where the practice of using one database, especially the use of relational database management systems, in supporting of Agile software development often suffers the

Liu J..
On Datastore Support for Agile Software Development.
DOI: 10.5220/0005106701830188
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 183-188
ISBN: 978-989-758-036-9
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

experience of very much a mismatch. Looking deeper, we believe that we have been doing this all wrong in our past projects where we would have just one database to support vastly different sets of data processing needs. The result is the requests of changing due to necessary software enhancement and resistance of changing due to stability considerations are constantly engage in this tug of war where neither side is winning.

In this paper, we argue that (1) many reasonably sized transactional based applications need at least two databases: one to support the basic customer facing business related transactional tasks, and one to support reporting needs; and (2) to effectively support Agile software development, the database handling customer facing tasks does not need to be a relational database. It is possible that the reporting database late can be modified to support data mining and data warehousing needs.

2 THE PROBLEMS

It is inevitable that almost all transactional based applications need some database support to make sure persistence data can be stored for late uses by the same or different components, or even a different application. In addition, when the information is needed, it can be quickly located among a large number of similar data. This persistence data store is generally accomplished through the use of files systems in the beginning of computer history and then switched to database management systems (Kroenke, 2013). For the past several decades, these databases have been relational databases (Kroenke, 2013). Because all important information for an application to operate properly has to be eventually stored in the database and then retrieved from the database, the database becomes a vital component of a software system. Because much of the code, both in UI and business logic are tightly coupled with the database schema, changing in schema often breaks the functioning features.

Looking back to several dozen applications we have participated in the past 20 some years, we realized that, although most of these applications have only one main supporting database, these databases generally service two very distinct sets of needs: store transactions and support reporting. We have attempted to use views and temporary tables to alleviate the negative effects of one part to the other and have very limited success.

In the past, being attaching with a large enterprise, a major milestone in our software projects has always been to complete the database design and declare database schema froze. The project then must activate Change Management process to contain negative effects on the necessary modifications to the database schema because modifications in other components may be absolute necessary to reflect changes in the database schema due to the tightly coupling nature among the business logic and UI database. Also the self describing characteristic of a database makes some simple changes, such as stop allowing a column to be null, be quite dramatic. Worse yet, once a software system is in production and becomes more and more popular, its database may need to provide data for other systems and databases. This is called the "worst-case scenario" in (Ambler, 2006). Changing the database schema in these situations can be a reasonable sized project by itself.

With more and more software teams adopting the Agile software development methodology, we are experiencing an even more frequent and urgent needs for schema changes. For several projects we have experienced, the schema may need to be changed several times within a sprint to accommodate different stories. Many tools, frameworks, and approaches have been developed to manage the inevitable changes, and some introduces new problems.

Ruby on Rails handles these changes by introducing the ActiveRecord package to automatically support the database needs. It also uses database migrations to move data from one database design to another. However, these auto-generated databases are good enough to allow an application to be demoed quickly; however, they are not well designed enterprise level databases and lack many of basic database objects such as indexes for fast data access and to support table join operations, stored procedures for processing data on the server vs. moving the data cross the network to be processed in the code, and views to hide the physical table structures. When the amount of data becomes large and number of concurrent users increases, these problems start to become more obvious. Another issue with the auto generated databases is security related. In a database centric design, which types of users have access to what database objects need to be well designed and carefully maintained. In an auto generate database, authentication and authorization related issues are, in almost all the cases, afterthought.

The approach suggested in (Martin, 2003) is to delay the decision of finalizing database designs.

When discussing the implementation of a use case, the author stated

> "Again, our predisposition to database may tempt us into thinking about record layouts or the field structure in a relational database table, but we should resist these urges."

Our field experience tells us that once we start coding, we likely need to store the data and pass the data to other components under development by other team members. Not storing the data, therefore not defining the data format or database schema, is not really an option.

A widely accepted approach is called "Database refactoring" (Ambler, 2003 and Ambler, 2006). It was first introduced in 2002 by S. W. Ambler based on the concept of refactoring for code. "Database refactoring" is defined as a simple change to a database schema that improves its design while retaining both its behavioural and informational semantics (Ambler, 2003). Code refactoring is very effective in enabling developers to evolve their code slowly over time and to improve their design, making it easier to understand and to modify. However, code refactoring and database refactoring are very difference in the effect of refactoring. The effect of code refactoring is mostly local. That is in the most situations, a refactoring activity retains the same semantics of modified code segment, at least from a black box point of view. We all know that an Object Oriented programming language like Java and C# can easily contain the implementation details of a method to the just that method. Database refactoring, on the other hand, is not simple because database schemas may be tightly coupled with code in applications it is supporting, other databases interacting with it, and code segments to support data migration and testing, just to name a few. The effect of some minor database refactoring can be huge and cause many unexpected situations. So, database refactoring should never be performed on a production database without thorough testing. In addition, working with experienced DBAs may be necessary to plan and perform database refactoring successfully.

There are other tools such as Liquibase that helps a project to manage database changes, especially for projects using Agile software development methodology. Still, it re-enforce the idea that changes are the necessary evil and need to be managed.

Generally an enterprise level customer facing application, such as Amazon.Com, needs to support three group of users: administrators and supporting staff, customers and CRM staff, and business analysts and management team. The operations performed by these three groups of users are very different and have very little overlap.

The administrators and supporting staff mainly need to monitor the status of an application and its environment such as CPU utilization, memory usage, storage performance, and network traffic. A subgroup of supporting staff may need to update the contents of a database such as adding new products or changing the quantities or prices of a number of products. Customers and CRM staff, on each access, need to be able to access and possible update a relatively small number of records quickly. Customers' actions may trigger changing of the state of the database. For example, a customer may purchase an item, which triggers the shipment process to start the steps of sending the item. Business analysts and management team perform very different sets of tasks that request very different support from the DBMS and data store. For example, business analysts may need to generate quarterly reports on sales figure on certain products. Such a report may need to access a large number of records from a number of tables, but business analysts generally do not need to change the state of a database.

Despite the fact where users of different group need very different support of data store needs, our past projects and many project across the world tend to use a single database, therefore a single database management system (DBMS) to fulfil all the database needs. This one-size-fit-all approach greatly restricts the software development progress of the project by disallowing or making it every expensive to change database schema because any change over database schema may results in rework in many parts of the application. Such changes are among main reasons software projects are late and over budget (Liu, 2009). Even with Agile software development, changing on database schema may have the domino effect of triggering re-work on many features and test cases that are considered as already completed.

Here is our dilemma, on one hand, to support rapid changes in requirements of software development, using or not using the Agile methodology, we have to allow frequent changes on the database schema; on the other hand, changes on database schema, such as a simple one as disallowing of nulls on a column may introduce many new defects when using a RDBMS such as MySQL or SQL Server 2012. We believe one

possible solution is to have two or more databases and with the one that supports transactional operations being not necessary relational.

3 THE SOLUTION

The truth is that for many enterprise level applications, the database supports are needed for two types of users: customers and business analysis and managers. Customers mostly utilize the transactional features of a DBMS at single ore a very selected few records level and hardly ever need to view aggregated results, while analysis and managers almost do the opposite -- they almost always view the database contents in aggregated views and hardly ever need to change the state of the database. After recognizing this, we cannot help but to ask why do we need to have one database rather than two or more? One explanation of this one-size-fit-all approach was historical. In the past, resources (hardware, software, and staffing) for a database were expensive. The situation is very different now. The workstations developers use to write code are sufficient to support most DBMS for development.

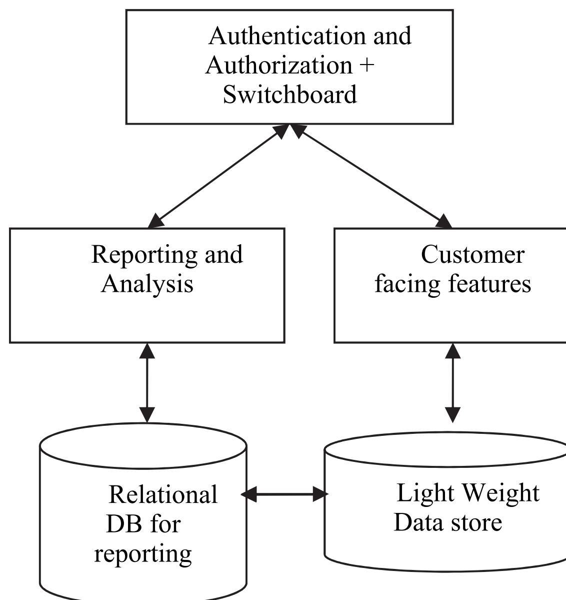

Figure 1: Applications have two supporting databases.

With open source, functioning DBMSs are readily available and affordable. With abundant tools, many with intuitive GUI, we may not need a DBA to create and maintain a database for us. We do not see any reason not to use two or more databases to support distinct sets of functionalities as shown in Figure 1. We call the database supporting customer facing activities the Transactional Database, while the other database the Reporting Database.

Considering the case of Amazon.com, a customer generally will view information in her account, search for items, complete an order, and to view comments from other customers regarding an item. Comparing to the amount of date Amazon.com stores, none of the activities require more than tiny amount of data. On the other, a business analysis many want to generate a report, say, to find out the top 10 most popular items in the last 10 days. That query may need to pull a lot of records and can be resource intensive. In the case of having one database, generating such a report may even slow down customer facing activities. On the other hand, a business analyst does not have much of needs to inspect individual records.

In the case the reporting database is used to perform Business Intelligence (BI) related activities, such as finding items that are associated with each other, the results of BI may need to be sent back to the transactional database. Otherwise, most of the time, the direction of data is to flow from the transactional database to the reporting database.

Since the transactional database hardly ever needs to support pulling of a large number of records, we believe the DBMS supporting such a database should be a light weight one that supports basic database operations (Select, Update, and Insert) well and does not even have to be a relational database.

4 WHY THE SOLUTION WORKS

Using multiple databases is not a new idea. Most data mining and data warehouse applications routinely load data from transactional databases through ETL, which is a mature technology and enjoy many effective tools. What we are proposing here is to start thinking of separating database needs between transactional activities and reporting activities at the very beginning of the system design and architectural stage so the changes in one database do not drastically affect the existing features supported by the other database. Such a separation can provide a very supportive platform for Agile software development because the developers can be much more freely in changing the database schema, especially the customer facing related database that generally require the support of transactional activities. Due to this separation, the

developers can change the schema of the transactional database with very little worry about affecting the reporting database. This freedom is not possible at all without the introduction of the second database.

Another benefit of developing systems with two databases is that since the transactions require fast date retrieval, data insertion, and record update for small amount of data, it may not be necessary to employ a fully featured database from a commercial management system (DBMS). We have participated in successful projects that use XML documents, MongoDB or other similar products.

MongoDB is an excellent example of an open source document-oriented NoSQL database management system. It provides a very different platform than the traditional RDBMS in terms of storing and retrieving information. For developers familiar with the RDBMS, MongoDB provides for indexes, dynamic queries, replication, and auto sharding as well as the retrieving, inserting, deleting, and updating of records. MongoDB is easy to learn and use because it supports, comparing with a fully functioning RDBMS such as Oregon 11g or SQL Server 2012, a very limit set of features--not a weakness considering its purposes.

Beside differences in terminology, one of the main differences between a database of MongoDB (called an instance) and a database supported by a RDBMS is that a MongoDB collection, similar to a table in a relational database, can hold any type of object, basically every record can be different in terms of format, data types of fields, and even the number of fields (we call columns in relational database terms). In addition, a record can contain arrays as a field's data type. This is MongoDB's approach of handling the traditional one-to-many relationship between two entities; as a result, it does not support the relational algebra's join operation and is certainly not relational.

It is out of the scope of this paper to discuss whether MongoDB's approach of handling one-to-many relationship is viable or not. Still, we'd like to mention that we do not appreciate the suggested handling of one-to-many relationship because MongoDB's approach, in our opinion, introduce complexity in searching on the "many" side. We also believe that MongoDB should consider adding support in joining two collections because join operations are used in retrieving data from databases all the time. After all, we cannot put the entire relational database into just one collection.

The benefits come from MongoDB's small set of features is obvious. Comparing for inserting 50,000 rows, MongoDB is 100 times faster than SQL Server 2008, a popular relational database management system produced by Microsoft (Kennedy, 2008). Even after considering that, during the test, SQL Server 2008 was accessed through LINQ to SQL as reported in (Kennedy, 2008), a 100 times difference is significant.

Retrieving data from a MongoDB is also faster than that of SQL Server 2008. The same article reported a SQL Server 2008 took 28 seconds to read out 50,000 records while MongoDB used only 10.4 seconds retrieving the same 50,000 records. For complex queries, MongoDB can complete 100,000 not so simple queries in 398 seconds. Doing the same takes SQL Server 2008 960 seconds. All tests described in (Kennedy, 2008) were conducted on a Lenovo T61 64-bit with a dual-core 2.8 GHz processor. The OS is on Windows 7, and all DBMS are 64-bit ones. We are in the process of conducting our own performance test and expect to provide our findings in our final version of the paper. After reading the detailed experiments given in (Kennedy, 2008), we believe SQL Server 2008's performance can be improved if proper indexes were added.

Note that, our experience shows that, for projects following the Agile software development methodology, schema changes on the transactional database can be frequent during the development phase, especially during the early stage of the development phase. Once a large number of features have been implemented, the scheme becomes stable. Once it is determined that the reporting related features is supported by a second database, the actual design of the reporting database can be pushed until the transactional database is relatively stable. As the result of separated databases, the reporting database not only is designed with a mutual understanding of its source database, but also is built on a more stable schema.

With proper design and architecting and adopting of the Layers of Data Abstraction concept, the reporting systems see the databases through external views, which enjoy immunity of changes in conceptual schema such as adding columns, tables, indexes, and views. It is this conceptual model generally needs to reflect changes in the schema changes in transactional database.

With increase in popularity on deploying customer facing application on the Cloud, separating the transactional database with the reporting database may become necessary for security reasons. The transactional databases are generally deployed with the application on the Cloud, which generally means it is not on the Intranet. For most enterprise

level security measures, authenticating users and authorizing accessing of resources on the Cloud using enterprise maintained user credential is unlikely due to security concerns. If we have only one database, we have to implement additional security measures to make sure the reporting database can only be accessed by business analysts and managers. Another sticky issue is that certain information, such as product costs and suppliers details, is considered as business secret and often is against company policy to be stored outside of the intranet. In that situation, we will be forced to bring data in the transactional database into the intranet to have a separate database. If we design our systems with two databases, meeting the security requirements becomes relative easy. Once the database is inside the firewall, we can use the existing authentication and authorization mechanism to control the access of the reports.

5 CONCLUSIONS

This one-size-fit-all approach of having just one database to support users of very different needs greatly restricts the software development progress by disallowing or making it every expensive to change database schema. With Agile software development methodology, we have to let developers change the database schema with certain level of freedom. To solve this mismatch between the database support and software development methodology, we propose to consider having at least two databases: one to support transactional capabilities and the other to support reporting needs. The database supporting the transactional operations does not even have to be relational, such as MongoDB.

We showed that due to the simplicity nature, the non-relational database can be faster than their relational counterpart. The separation of the transactional database and reporting database provide easy support of data mining and data warehousing extensions on the reporting side, provide much great freedom for the developers to use the best database support to implement customer facing features without worry about affecting the reporting side functionalities. Last but not least, because the reporting database is independent, it can be hosted inside the firewall while the transactional database is deployed on the Cloud.

ACKNOWLEDGEMENTS

We would to thank Western Oregon University for their support in our research and writing of this paper with their research grant JL2014.

REFERENCES

Turk, D., France R., and Rumpe, B., 2002. Agile Software Processes: Principles, Assumptions and Limitations, Technical Report, Colorado State University

Kennedy, M., 2008. http://blog.michaelckennedy.net /2010/04/29/mongodb-vs-sql-server-2008-performance-showdown/

Sommerville, I., 2006. Software Engineering", 8 Ed, Addison-Wesley

Liu, J., Liu, F., 2009. Factors contribute to high costs of software projects, the 2009 International Conference on Software Engineering Research and Practice, July 13-16, 2009, Las Vegas, USA

Martin, R., 2003. Agile Software Development, Pearson Education, Inc., Upper Saddle River, NJ

Ambler, S., 2003. Agile Database Techniques, John Wiley & Sons

Burns, L., 2011 Building the Agile Database, Technics Publications

Ambler, S., Sadalage P., 2006. Refactoring Databases: Evolutionary Database Design, Addison-Wesley Professional

Kroenke, D. and Auer, D., 2013. Database Processing: Fundamentals, Design, and Implementation, 13th ed, Pearson Prentice Hall, Upper Saddle River, NJ

.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS