Applying Event sourcing in a ERP system: a case study

Pedro R. G. Vasconsellos
School of Computing and Informatics
Mackenzie Presbyterian University
São Paulo, São Paulo, 01302-907
e-mail: prgoes@gmail.com

Vinicius M. Bezerra
School of Computing and Informatics
Mackenzie Presbyterian University
São Paulo, São Paulo, 01302-907
e-mail: vinicius.bezerra@mackenzie.br

Calebe P. Bianchini
School of Computing and Informatics
Mackenzie Presbyterian University
São Paulo, São Paulo, 01302-907
e-mail: calebe.bianchini@mackenzie.br

***Abstract*—Most software can be seen as a set of business rules that are executed when triggered by an internal event or user input. Usually, these business rules are designed to process the input data and change the software's internal state match this event. As a consequence of such design, software usually does not have reliable means to keep memory from its past states unless specifically programmed to do so. Even when such capabilities are built into the system, they are often too detached from the system's rules to be useful or they are built only in particular points of the system selected as relevant. This article discusses Event Sourcing, a software architecture design that organizes software's classes in such a way to provide a native memory of its past states, enabling the system to even replay old computations at will. We also present a case study where this architecture is applied in the development of an ERP System. The architecture was particularly useful to the development team as it dramatically increased error traceability.**

***Keywords*—event sourcing, ERP, software design, software architecture**

I. Introduction

Object-oriented software is usually composed of at least three specialized layers [1]: presentation, business and data. The presentation layer is responsible for presenting information for a user and collecting general input, the business layer focuses on modeling the relevant business rules for achieving software's goals and the data layer is the one that saves and retrieves operational software state from a reliable persistence mechanism, usually after a process known as Object Relational Mapping [2].

The business layer is where most (if not all) business rules are concentrated and its interface can usually be split into commands and queries [3][4], where commands are methods that alter the current state of an object and queries are the ones that retrieve this state for other interfacing objects. This way, software emulating a business world process will change its current internal state over the course of operation and frequently inform the interested parties when queried about it.

An important aspect to consider from this architecture is that new software state will effectively erase the previous one unless it was specifically programmed to do so. When a command is received, it will be intended to change the current state of the objects that will be loaded by the data layer and later overwritten in the persistence mechanism. As such, software programs usually do not have any kind of its internal state history although in many cases this information can be very relevant to be analyzed.

The lack of internal state history memory in software makes it hard to understand past computations. From a basic possible software operating cycle (receive an input, load a previous state, change the state and save the new one), one can infer that a computation's results depend exclusively on three aspects: the software's objects current state, the received input and the business rules programmed into it. This means that in order to understand a computation done in the past one would need all this information as it were back in the day the computation was done. Analysing the 3 pieces of information needed, the input is very much tied to the computation itself so, if one needs to study a past procedure, the input is mostly known. Next, the business rules as they were programmed in that time is an issue mostly covered by a source code versioning system. This leaves the whole software state as the only aspect hard to obtain in the future since any operation after that particular point in time will most likely have overwritten any relevant data to replay it.

This article presents a case study where Event Sourcing was applied in the refactoring of an ERP system. The main difficulties and the advantages of this change in the system architecture are discussed and analyzed.

In the next section, the literature on the internal state history issue is presented and the alternative solutions to Event Sourcing are analyzed. Then, Domain Driven Design and Event Sourcing patterns are presented. Finally, a case study where presented when Event Sourcing was applied in the process.

II. The internal state history loss problem

The fact that software, in some level, lacks history about its past states affects different areas for different reasons. Two of such areas are production code debugging and analysis and business intelligence.

A. *Production code debugging and analysis*

One of the main challenges in production code debugging is error traceability and reproducibility [5][6]. It is known that it is nearly impossible to guarantee a software program

to be free of bugs despite the different valid attempts of minimizing bugs in early development stages such as Test Driven Development and other quality oriented techniques [7]. However, this means the problems that are left in code will usually be the the hardest to find [8] and fix. One of the reasons fixing production bugs is very hard is because one has to understand what was happening inside the code when a given error was detected. Many times, error reports will end up with a "Cannot Reproduce"/"Works for Me" reply from the investigating party, as shown in various studies considering bug lifecycle [9][10], that will lead nowhere as far as fixing the problem goes. In other words, many times, an error will be detected in production environment by a user, reported to the proper quality assurance entity and have its correction prevented because the error can not be reproduced. One of the reasons for that is because it is really difficult run debugging routines in production. During the software development process, even in Quality Assurance stages, the environment is usually very controlled with preconditions building the general state to the testing point, test steps well documented and relevant outputs recorded in logs. Such conveniences are rarely available once the program reaches production stage as they impact the system performance and size of storage required by logs. To be able to achieve reproducibility in a software computation three items are required:

- The Source Code Version. One will need the exact source code version in which the error was detected in order to reproduce the same steps the production code executed. This is solved by the usage of any source code versioning system.
- The Input. The error happened as someone or something - in the case of automated interactions - tried to perform an action, supplying an input to the system. This is usually information known by the reporting party.
- The Environment State. As software consist of a set of implemented rules running on some software state, the state itself is a critical part of the computation while being the least trivial to retrieve as it is constantly being overwritten.

One common answer to production debugging in software comes in the form of logging and it consists of key statements inserted into specific code positions in order to gather and record information about its current operation and state. Among others, typical places for logging will usually be the beginning of a method (collecting input data) and exit points, both regular and exception ones (collecting output and error data). Although largely used, the logging option does have some significant drawbacks.

First, software logs don't give the developers any tools to actually rebuild software state on their own. They might provide with software state information but actually rebuilding the state will probably depend on multiple log entries that won't be easy to gather and then a mostly manual process must be carried assuming the correct information was found.

Another problematic aspect of logging is that a log is only as good as the information it collects [4]. The review of the state of art shows two approaches that try to solve this problem. The first is an empirical technique to enhance log generation, which heavily relies on code quality and sub sequential follow up and inspection to guarantee logging quality. The second is a tool to automatically analyze source code and improve data collection capabilities of software logs [11] that can greatly help leveling the quality of the generated logs, but debugging production code still remains an issue.

B. *Business intelligence*

The second area affected by the lack of internal state history is Business Inteligence. In the past decade, Business Intelligence (BI) has risen as an important decision support tool for companies around the globe. Nowadays, it is hard to imagine a successful company not taking advantage of the BI tool set [12]. These applications largely use historical data in order to generate reliable predictions for future operations. In order to work with this historical data, BI software rely on collecting data as it is generated and transforming it for its own storage and purposes in a process called ETL. One of the drawbacks of this approach is that one will only have valid BI historical data on that information which was predicted to be needed and thus included in the whole collection and transformation processes.

On a high-level, any BI architecture will work pretty much the same way. In a first tier, there are the various Data Sources of interest where the raw data the company intends to analyze lie. The second tier (Data Movement, Streaming Engines) is where the collection of this data into the BI engine will happen and is normally comprised of two kinds of process: the Extract Transform Load (ETL) will query the tier one data sources periodically for data and transform it into a more analysis-friendly format and the Complex Event Processing (CEP) will continuously monitor a data stream looking for predefined patterns interesting to the business in question. Both the ETL and CEP will push its data into the third tier (Data Warehouse Servers) where BI data will actually be analyzed and stored. In the fourth tier various Mid-Tier Servers will be present to query and organize the collected data in different ways. Finally, the fifth tier will contain the Front-End Applications to present the data to the business in various forms.

One of the critical aspects for a successful BI system is the presence of a cohesive, organized and historical data warehouse [13] [14]. If one does not correctly predict which data would be needed, in which interval and how to correctly organize it during the ETL, one might end up with a large amount of pointless data.

The main point here is that the ETL process is based on a static model. One will watch the general state of the data and take predefined snapshots of that state. If one changes its mind, the new way of thinking will only be applied to subsequent data extracted. As these analyses rely heavily on the amount of data collected for effectiveness anytime a change is needed, it will take sometime before new ones can be made.

III. DOMAIN DRIVEN DESIGN BUILDING BLOCKS

Before discussing Event Sourcing (ES) as a technique, it is important to define three of the Domain Driven Design building blocks [5], which are used on ES and their definitions. Those concepts are Entity, Value Object and Aggregate.

Entity and Value Object are types of objects that are used for defining the domain model. An Entity is an object with a personal identity. This identity will never change throughout its entire lifecycle even if all its internal attributes and state change drastically. It should be noted that the concept of lifecycle here mentioned is broader than that of the object - which lasts from instantiation to destruction. This lifecycle is the Entity's, which continues even if its representing objects are temporarily destroyed and stored in a database. A Value Object, on the other hand, lacks this sort of identity. As the name suggests, one is only worth for its internal value. When a concept is modeled as a Value Object, the instance in question is irrelevant for its use and, if you have two instances of a Value Object with the same internal values, for all purposes they are the same and can be used without differentiation. Value Objects should also be treated as immutable.

The final construct, Aggregate is a cluster of Entities and Value Objects that model a larger concept. Modeling such concept in a single class would break the Single Responsibility Principle [4] and most certainly be accounted as a Code Smell and be candidate for refactoring [15]. Still, despite being modeled as a group of classes, it is highly convenitent to identify such concepts as a single construct and this idea was called an Aggregate.

When discussing an Aggregate a few important related concepts must be defined. The first of them is the Aggregate Root which is the class that will hold its identity and all its inner components (be them other Entities with Identities or Value Objects).

The second point to be noted in an Aggregate is that the Aggregate Root is in charge of Invariant Control. This concept handles the idea that, within an Aggregate, there can be rules that span more than one inner component and they must be checked and enforced by a central component and componente which should be the Aggregate Root.

With these concepts defined, we can move on to the discussion on Event Sourcing itself.

IV. EVENT SOURCING

Event Sourcing (ES) was first proposed as a pattern by Vernon Vaughn [16] while studying and extending Eric Evans' Domain Driven Design (DDD) [5]. As it stands, ES presents a very viable alternative to fix the aforementioned internal state history loss problem. Although it is theoretically possible to work on ES on different software contexts, this article will only analyze the application of ES on object-oriented software, specifically those created based on Evans' Patterns and Principles presented on his book [17]. As a specific derivative of Evans' DDD language, this article will discuss his patterns and lessons learned during the ES implementation.

The concept behind event sourcing is a simple one: what if, instead of storing the software state, one would store the actions that led to this state? In a simple analysis, one could relate this change of perspective to a Turing Machine. Having universal computation power, a Turing Machine's computations consist of the same basic principles as modern software and since Object Oriented languages, such as C#, are usually Turing Complete, one could analyze ES behavior from a Turing Machine's perspective. In a simple model, a Turing Machine will have:

- A state register, which will mark the machine's current internal state.
- A tape, filled with a finite alphabet.
- Production Rules, which define: if the machine is on state X_1 and reads the symbol Y_1 from the tape, it will write symbol Y_2 on the tape, move it in direction D and assume state X_2.

Now suppose one would want to interrupt a Turing Machine computation to restart it in the future. One way of doing it would be to store the Production Rules, the current tape and state register. When attempting to continue the computation, simply reloading the Production Rules in to the machine, using the same - or an effective copy - tape and setting the State Register to the remembered state would make the machine continue from the exact same point. That is a lot like what we do in modern software. The business rules are pretty much always "saved" in the form of the executables and libraries. The internal state is what one saves to the database, transforming software objects into a relational structure.

Although this is the most usual way of running this operation, there is an alternative. Since Turing Machine is always deterministic (a non-deterministic one can be converted into a deterministic counterpart) executing the same production rule over the same state-symbol tuple will always yield the same result. This effectively means that, instead of storing a complex mid-computation state, one could store a simple initial state and the rules, and from that point the computation history would follow the same steps again. This same logic could be applied to modern software. Instead of storing the whole software state from one computation to another, one could store every "action" or "event" that triggered a state change. That way, when one would need the software's current state, one could simply replay the events in the order they occurred and obtain the exact same state again. By going through this approach, though, one would be able to discover the state of the software program in any given point in time, simply by replaying the events only until that moment. Vernon [16] called this technique Event Sourcing.

In an ES implementation, a DDD-style Domain Model [18] contains Entities representing a strategic abstraction of the real world domain being represented. These Entities will then expose commands, i.e. methods with no return value intended to alter the Object's internal state, as proposed by the Command Query Separation principle (CQS). Once a command is triggered by an external actor [19], instead of

directly updating the Entity's internal state, it should fire and register an internal Domain Event [5] object which will then go through an Event Handler that will be responsible for updating the Entity's state. To stay in accordance to the principles of DDD this Domain Event can easily be in full compliance to the Ubiquitous Language (UL) [5], by using terms of the UL reflecting the action it represents. Figure 1, drawn as a free form model, illustrates this process.

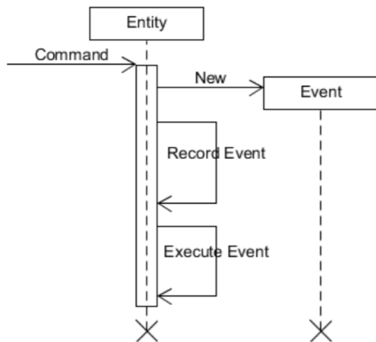

Fig. 1. Event sourcing processing an Entity command

As an example, a sales system is presented. The main concept in this system is an Order. This Order will be composed of several Line Items, each of them representing a Product and a Quantity included. A way of modeling this would be having two distinct Aggregates (Order and Product) with Order being an Entity and the Aggregate Root of the Order Aggregate and Line Item being another Entity but a member of the same Aggregate. Product then, is the Entity Aggregate Root of the Product Aggregate, limiting the kind of reference Line Item can hold to it. Quantity is also a part of the Order Aggregate being a Value Object and not an Entity. Figure 2 contains a free form model representing this scenario.

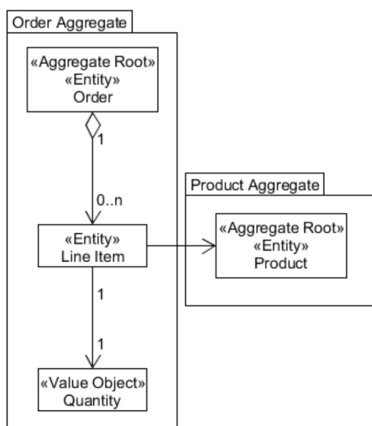

Fig. 2. Order and Product Aggregates example

This conceptual model could be developed in both a regular and ES way, but the difference to their coding would be drastic. The pseudo-codes presented on Listing 1 and 2 show the difference in organization and spirit of both approaches.

Listing 1. Common Coding Approach.

```
1 class Order
2 {
3   public void AddItem(Product aProduct, Integer
        quantity)
4   {
5     this.Items.Add(new LineItem(aProduct, quantity))
          ;
6   }
7 }
```

Listing 2. Event Sourcing Approach.

```
1 class Order: DomainEntity
2 {
3   public void AddItem(Product aProduct, Integer
        quantity)
4   {
5     this.Trigger(new LineItemAdded(aProduct,
          quantity));
6   }
7
8   public void OnLineItemAdded(LineItemAdded event)
9   {
10    this.Items.Add(new LineItem(event.product, event
          .quantity));
11  }
12
13 }
```

At a first glance, it should be noted that ES involves a lot more coding. The simple method *AddItem* (lines 4-7) from Listing 2 will be handled all in itself in a Common Coding Approach, but will need four support methods to perform its job in ES: *OnLineItemAdded*, *Trigger*, *RegisterNewEvent* and *ExecuteHandler*. That way, at a first sight, ES presents one drawback on making code bigger and more complex. Despite the disadvantage, let's take a look into the methods and classes that will take part in an ES routine:

- *LineItemAdded*: This is the class that will carry the event payload, i.e. the data needed to fulfill the event at a later step. This class is mostly a data bag, carrying support information (such as event date, see the implementation analysis section for further details) and the actual payload for the event, which in this example would be the Product and Quantity involved in the Item creation.
- *OnLineItemAdded*: This method is the Event Handler for the *LineItemAdded* Domain Event, which means it will do most of the work the original method in the common coding approach did, i.e., actually performing the state-changing task.
- *Event*: Base class for all Domain Events.
- *Trigger*: The main method (common for all Event Sourced Entities, present in the base class: DomainEntity) to trigger any domain event from inside an Event Sourced Entity. It calls RegisterNewEvent and ExecuteHandler.
- *RegisterNewEvent*: Inherited from DomainEntity, records, on some sort of internal structure, which new events have been triggered since the last time the class was loaded from previous events.
- *ExecuteHandler*: Inherited from DomainEntity, looks for an appropriate handler for a given Domain Event and will invoke it immediately. The Event/Event Handler mapping

can be created in a few different ways which will impact code readability and performance and are discussed in the implementation section.

From the pseudo-code and element descriptions, it is noticeable that, despite the fact that an Event Sourced command will involve more coding than a standard one, much of this extra code will be a one-time effort common to all Event Sourced Entities. The actual extra code is restricted to the Domain Event and its Event Handler. Also, it should be clear that there is no delay between the issuing of a command and the command effect to be observable from outside the Entity (i.e. by the time the *AddItem* method finishes and execution control returns to the calling method, the effect of the command will have taken place in both ES and Common strategies).

Another aspect that should be clear is that from a client point of view, operating on the Event Sourced Entity does not represent any change in behavior or programming style. Both queries and commands issued upon the Entity will behave and look like exactly as a regular Entity would. The aspects, though, which will present more changes in the process are both persisting and loading such an Entity.

Since every command issued upon the Order Entity will generate a new Event that will be registered in some internal structure by means of the *RegisterNewEvent* method, it will be possible to query which Events happened in an Aggregate since the last time it was loaded from the database. That way, when a command is issued to the Data Layer to Save the Order Aggregate, instead of mapping the Order Aggregate's fields to a set of Tables or other paradigm's storage structures, the Unit of Work [18] will gather the Events issued since the Entity's last load and will store them into the Database. This database will then be essentially an Event Store, recording primarily every single Event from every Event Sourced Entity in its context [20]. As an example, Table I shows how a portion of an Event Source database could look like. For this example, assume a unit of product 927 costs 10 units and product 478 costs 100 units. As it can be seen, the sequence of events is quite easy to read and understand. In the first line a *Customer*, coded *102*, had its *Purchase Limit Reduced* to a *New Limit* of *100* currency units. One minute after that, *Sales Rep 8* placed a new *Order* to this *Customer 102* with an *Order Number 507*. One minute after, this *Order 507* had *3* units of *Product 927 Added* to it. One minute further, the item containing *Product 927* had its quantity changed to *4*. Finally, fifty-seven minutes after, *Customer 102* had its *Purchase Limit Increased* to a *New Limit* of *1000* currency units.

Now suppose this *Sales Rep 8*, at some point later in the 18th, calls the vendor of this particular software complaining that, after adding *4* units of *Product 927*, he could not add *2* units of *Product 478* and thus could not complete his sale. The person investigating the issue, aware of the program's business rules, would ask whether *Customer 102* has enough *Purchase Limit* to have that *Order*, since the *6* units would total around *240* currency units. As we are past *3 PM* (15:00) in the 18th, the *Sales Rep* would verify that indeed *Customer 102* has got a limit of *1000* currency units and a new set of investigations

TABLE I
EVENT SOURCED DATABASE EXAMPLE

Event Id	Time	Entity Type	Entity Id	Event type	Event Data
(... older data ...)					
100	2014-10-18 14:00:00	Customer	102	Purchase Limit Reduced	New Limit = 100
101	2014-10-18 14:01:00	Order	507	Order Created	Customer = 102 Sales Rep = 8
102	2014-10-18 14:02:00	Order	507	Line Item Added	Product = 927 Quantity = 3
103	2014-10-18 14:03:00	Order	507	Item Quantity Changed	Product = 927 New Quantity = 4
104	2014-10-18 15:00:00	Customer	102	Purchase Limit Increased	New Limit = 1000
(... newer data ...)					

would take place in attempt to either find a bug - in this case, a non-existing bug - or to understand what actually happened, which could take a long time. Instead of that, using the Event Sourced database, the investigating person could simply replay the database in a local copy up to *2014-10-18 14:02:01* (or even just 1 millisecond after *14:02:00*), and actually perform - and debug - the action to add an item of *Product 478*, with *2* units, to *Order 507*. He would then see that, by the time the *Order* was being placed, *Customer 102* indeed did have a lower *Purchase Limit*, which was not enough to fulfill this particular *Order*.

The second relevant aspect that should be noted in this database extract is that every Event of every Entity is placed in the same structure - in this case, a table. The reason for such decision is that when time has come to rebuild the system from its temporal data, having Events scattered over multiple sources would present a great deal of issues. Suppose on the example above that Order Events and Customer Events either had its own table. When attempting to replay Order Event number 101, one would need to check if every other Event Structure had its events replayed up until *2014-10-18 14:01:00*. This verification of other sources would need to be done on every Event of every Entity greatly increasing the number of queries needed for a rebuild. If Events are all stored in a single timeline, though, no dependency checks are needed and one can simply replay the Events in the order they appear. The drawback of this decision is that all events would have to fit under the same schema. To solve this problem, the Event table contains a free form Event Data field that can hold some serialized form of the Event-specific operational data.

With the basic structure of an Event Sourced command and the storage structure for the Events out of the way, we should look at how an ES system can rebuild its entities from the Event Storage. The easiest way to rebuild an ES Entity is to get it from its id. The Repository [18] responsible for the operation will then simply query the Event Storage for every single Event with that particular Entity Id and rebuild the Entity from there. On the side of the Entity, one should have a constructor that accepts a Collection of Events as its parameter and rebuilds the Entity from that Collection. The pseudo code is shown at Listing 3.

One might notice this constructor goes straight to the event handlers ignoring the original commands and the event registration. Of course, the original commands were only

responsible for keeping invariants checked and triggering the event. If the event has already been triggered, it has passed the invariant checking step not needing to be evaluated again. Also, this way the event registration is not triggered again, meaning this particular instance of the Entity will have a clean record of recent events, thus, making the job easier to save it when time comes - every event registered by the RegisterEvent method will have been inserted after the load from history, meaning it is not already in the database and should be persisted.

Listing 3. An example of a constructor in an Entity type.

```
class Order
{
  public Order(List<Event> history)
  {
    foreach(Event e in history)
    {
      this.ExecuteHandler(e);
    }
  }
}
```

A question that might arise regarding ES is how well the system performs with an entity that changes a lot during its lifetime, i.e. an Entity that will have a large amount of history events to be rebuilt from. In such cases, the ES system can be expanded to support the concept of Snapshots. The idea of a Snapshot is to capture an Entity in its entirety at a given moment of time, thus removing the need from the system to replay every single event in the event stream - only those events that happened after the Snapshot need to be replayed.

V. Case Study - Event Sourcing at Liera

Liera is an ERP being developed in a logistics-focused company during the operation of a previous ERP system. The new ERP is evolving replacing module by module of the old one during the company's day-to-day operations, and integrating with the old ERP's modules that are still live in production. That scenario, aggravated by an ERP's natural complexity and Brazil's complicated tax laws - with various complex scenario-specific tax rate changing - made in-production debugging a critical aspect of development. Another important issue to be dealt with was regarding answering questions about past operations like why that particular tax rate was applied to that particular order or what happened with this order being rejected three times by logistics team before being shipped to the customer. All these questions invoke knowledge about past software rules and parameters usually unavailable in a regular object oriented piece of software. With all those aspects into play also being pushed by past issues with log-based debugging, Event Sourcing quickly rose as the option to go for the team.

A. Database Selection

Despite all the positive aspects pointed to adopting Event Sourcing into the project, quite a few issues were still in the way for the development team. The first of them lied with the choice of database, where ACID/Non-ACID and the usage of well-defined/flexible scheme are some of the analyzed features.

Although a hot topic, the performance difference between them was left out of discussion because too many factors come into play when deciding the best performance achievable in a given scenario. As pointed out by [21], MongoDB performance is superior in inserts, updates and simple queries while MSSQL is faster when querying non-key attributes and aggregate queries. In another study, [22] concludes that, on average, MongoDB would be faster than MSSQL but looking at the data it is easy to see the performance varying depending on the number of operations performed.In a third article, [23] analyses, without benchmark data, the factors which might improve NoSQL performance and concludes some of them to be absent on different NoSQL implementations and able to be replicated on SQL ones, that way, the performance would be better analyzed from case to case.

Adding to these articles, Event Sourcing also brought a few concerns regarding its own performance as a model. First, the heavy usage of BLOBS in a MSSQL implementation would probably mean a performance hit to that of the average MSSQL system. Second, the fact that an Event Sourced system does not run update operations on the Event Store - new events are pushed into the store, never to be changed or deleted - would certainly have some level of impact on the performance balance between implementations. Finally, the system being developed would not be possible without some level of transaction control, which would hinder an eventual MongoDB implementation. Considering all those factors, the performance indicator was kept at large through most of the decision process.

With the performance factor out of way, the database decision was reduced to native transaction handling on MSSQL's favor and schema flexibility pointing to MongoDB. In the end, the team considered the BLOB overuse risk to outweigh the manual transaction control to be performed on MongoDB, so the NoSQL implementation was given a go.

B. Concurrency

The transaction control implementation took advantage of the fact that the Event Store is an Insert-Only Collection and new Events are only appended to the end of the stream. The only conflict possible in this scenario is to have two events altering the same entity being written at the same time by two different sources, which can be handled in a basic scenario of Optimistic Concurrency. To deal with this, both the base Event Class and the base EventSourced Entity Class of the system were given an Integer representing its Event Version. The EventSourced Version value would be that of the last Event either replayed during its reconstruction or the one of the last Event generated through a command on its current instance. The Event Version value would be one plus the current Version of the EventSourced Entity where the Event was generated. As the Event also carries the SourceId GUID of the Entity it belongs to, on the Database side the pair SourceId - Version of the Event Store documents was made unique. Since Events

should always be inserted in the order they were generated, two concurrent UnitOfWork objects would not be able to push Events that fail the Optimistic Concurrency constraint into the Event Store.

Figure 3 illustrates this concept. In the first step, two concurrent processes load the same entity 101, which is currently in version 10. Then, in step two, they both generate new events on their local instances of the entities, creating events versioned as 11. After that, in step three, process 1 saves its changes to the event store, pushing its new event into the stream. Finally, when we reach step four, process 2 attempts to do the same, it meets an Optimistic Concurrency conflict as the Event Store already possesses an event with Version 11 for Entity 101. This shows process two holds an out of date version of the Entity and its save operation must fail, but the Event Store integrity will be safe.

Although this was a nice safeguard for the Event Store, it still would not cover all the possible concurrency damages. First, multi-entity transactions would not be protected at all. One UnitOfWork could be committing two Entities at the same time, and a concurrent one could be committing a single Entity that would conflict with the other transaction (as in the example shown before). If the second commit takes place first, the first UnitOfWork would make an inconsistent commit of a single Entity, since the other would fail through the Optimistic Concurrency safeguard. Another possible failure would be an UnitOfWork committing two Events of the same Entity and, between the actual write operation of the first and the second Event to the database, a very fast concurrent UnitOfWork reads the updated Entity - thus having the Version number of the first written Event of the other UnitOfWork - and writes a single Event of its own. The second Event of the original UnitOfWork would fail, and the first commit would then be inconsistent.

Figure 4 illustrates this scenario. In the first step, Process 1 already has Entity 101 loaded, with original Version 10 and two new Events, 11 and 12 generated. In step two, it starts its save routine and successfully pushes Event 11 into the Store. At the same time, Process 2 loads Entity 101, now with Version 11, even though the saving routine from Process 1 is not over. After that, in step three, Process 2, which is faster, generates an Event versioned as 12 on Entity 101 and successfully pushes it into the stream. Then in step four, when Process 1 tries to finish its original save operation, it reaches a conflict on the Event versioned as 12. This is a major concern since at this point the integrity of the Event Store is completely compromised. First, Entity 101 Version 11 should never actually exist, since it was never intended to be saved on this state (Process 1 intended to save Entity 101 as Version 12). Second, the Entity generated from Process 2 is completely invalid since it is based off a state that should never exist in the first place. From these scenarios, it is clear that the whole group of Events being committed - whether from a single Entity or not - need to be committed as an atomic transaction.

To deal with this scenario, a Two-Phase Commit and Two-Phase Locking UnitOfWork was designed. The designed locking mechanism was simple and took advantage of the design decision that every Entity had a GUID as its primary ID. That way, a simple MongoDB collection where the field LockedEntity (GUID) was unique held a record for each entity locked. A few other support fields such as the locking transaction and lock creation time were added to help the process of eventual lock cleanups, but the main process consisted of the UnitOfWork writing an entry in that for each Entity being committed. Being successful on that meant the locks were achieved. Failing, meant one or more entities could not be locked by this transaction so the commit would not take place. As for the Two-Phase Commit, the Preparation phase would begin with the Entity locking process and continue with a version verification of every Entity present in the transaction. With every Entity version validated and locked, the Commit Phase would follow, pushing the new Events into the Store and finally releasing the locks obtained. This design, of course, means committing a large number of independent Entities into the database could affect performance by a large amount, but this wasn't considered a problem since DDD design goals aim to avoid multi-aggregate committing, thus, minimizing this issue.

C. Entity Searches

The second overall issue to deal with the Event Sourcing implementation is Entity searching. In a regular development environment, it is usual to answer questions such as Find the Orders made by Customer X, which usually means matching a Order record against its Customer field. In an Event Sourcing development though, these fields are not organized in an intuitive way: an Entity's data will be scattered between the many events that constitute this Entity's state. To solve this, a few approaches were considered by the team. The first one was a Command Query Responsibility Separation (CQRS). In a CQRS environment, one maintains two distinct models to work with. The command model is responsible for receiving all of the software commands and since commands are usually issued through the Entity's ID, it is simple to handle them in an Event Sourced Event Store. The query model then is responsible for answering for any queries ran against the system and is generated from the commands issued. That way, there is a possible delay between a command being issued and its result being reflected on the query models, but the query models are able to be tailored for specific query needs, having aggregations and other concerns addressed during their generation, making them very efficient in runtime. Although it may seem very well tailored for an Event Sourcing approach, since the Event Store is almost a natural representation for a command model and query models are very easily generated from the Events. Despite these advantages, this strategy was not chosen one because the team felt they were not sure how the concurrent models and update delays would work in production environment.

The second approach considered for this development was a very similar one, but instead of a full segregated query model, the model used for queries would be composed of

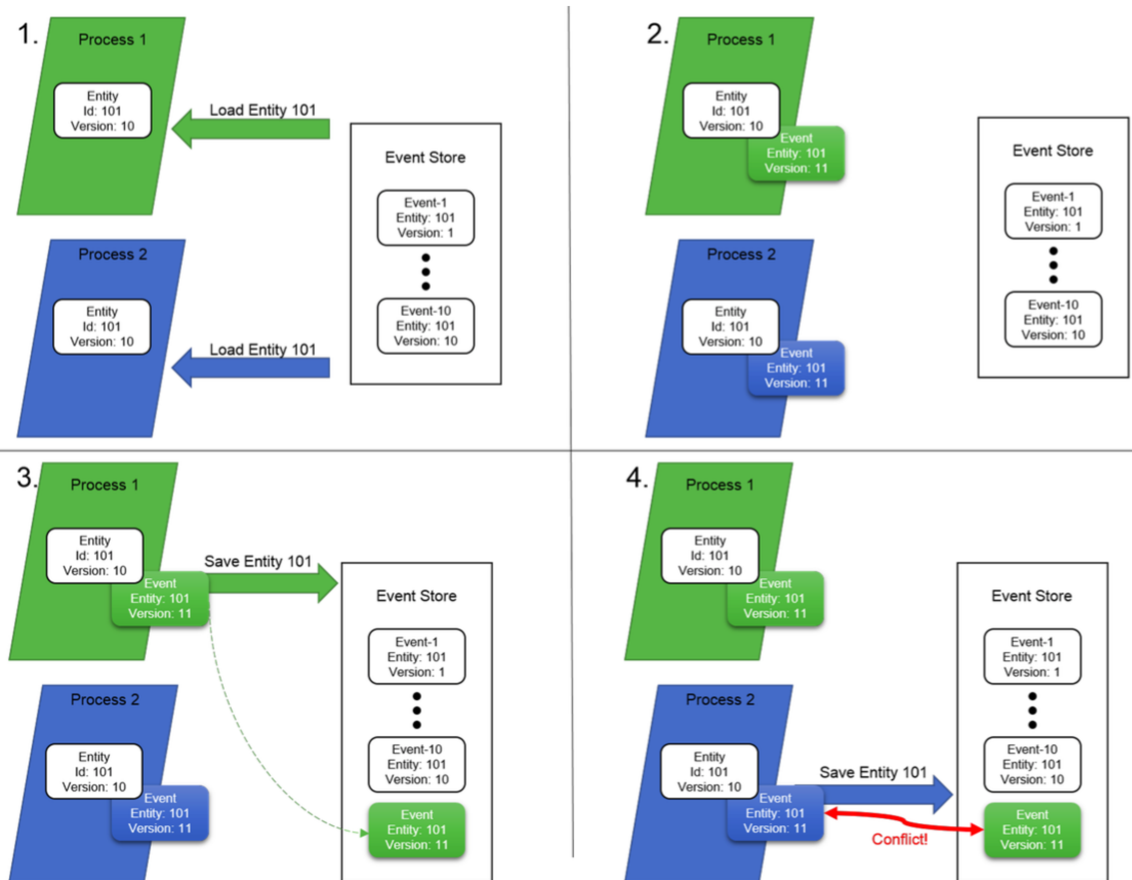

Fig. 3. Optimistic Concurrency Conflict

the snapshots of the various Entities mapped in the Event Store. While certainly a simpler approach than the CQRS one, the delay generating the read model is still present and this approach was discarded.

The third approach relied on the fact that, using MongoDB, the Event Store could be actually queried for event data. This approach allowed to issue a query looking for events from entities of Order type that contained the field Customer having its value set as ”X”. The approach ended up being discarded as well, after taking in consideration that there wouldn't be an effective way to index all the possible searches, once the Event Store grew on size, finding a single document would probably become too slow.

The final approach considered and chosen for this scenario came from the third one where, instead of querying directly into event data, one could make the Entities publish a list of *queryable* keywords to be searched for in the future. When an Event is about to be written to the database, it would incorporate the current reading of this keyword collection into an internal keyword array. Taking advantage of MongoDB's array field indexing capabilities, this field, common to every Event on the Event Store, could be queried and indexed freely.

Despite being considered the most attractive for solving the Entity searching problem, the keyword approach was not free from issues itself though many of them were present in the other approaches as well. One problem is the possibility of false positive results. For example, consider the Student Entity publishing the Student's Name as one of its keywords. If one day in the past the Student's Name has been Bob, the events of that era would carry the Bob keyword in them. If later this particular Student has been renamed as Robert a search for events containing Student as their Entity Type and Bob as a keyword would find the older events of this Entity making it a candidate result. Only after the Entity's full rebuild cycle it would be verified that this Entity is in fact not a suitable result. This issue could possibly scale very badly if present on a *queryable* field too frequently updated. That way, the amount of false positives could increase very fast forcing the reconstruction of many unneeded Entities crippling search performance. To help mitigate this issue, a secondary service was designed to search through the Event Store for predefined Events that belong to Entities prone to generating false positives in queries. This service would update the Event's keywords to reflect the current state of the Entity, effectively removing false positives until the next Entity update. Although this action violates the concept of an Event Store being a push-only stream, with no updates or deletes being made on Events, this sort of support information

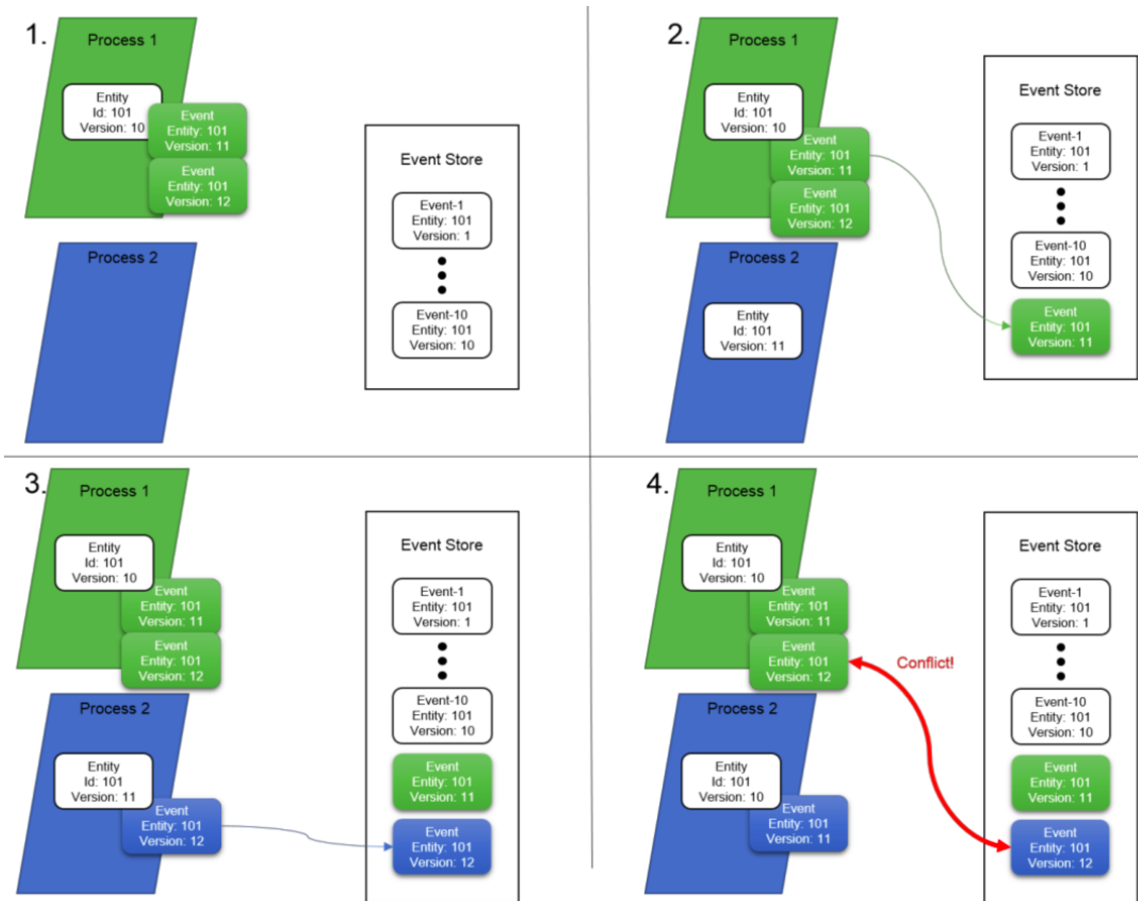

Fig. 4. A more complex Optimistic Concurrency Conflict

update seemed to be safe, since it does not alter the actual event data. In addition, as the rest of the system abode by the push-only principle, this update routine would not even have to worry about concurrency and other issues. The other issue found was that the amount of keywords each Entity published had to be kept relatively small otherwise the Event Store's size would quickly grow out of proportion. Since every event pertaining to a given Entity carry every keyword published by said Entity, this can quickly become a huge overhead reducing the ratio between an Event's true data and its support data wasting precious storage. This restriction would of course conflict with the freedom to query any desirable field from an Entity so a compromise was made with the decision to use the keyword approach for delay-sensitive data and operations and use the second approach, the snapshot based search for the not so delay-sensitive ones. The responsibility to decide which search mode to use would rely on the Entity when choosing which keywords to publish and the respective repository when deciding how to query the database to find its results.

D. Code Versioning versus Events

The final aspect to be considered in the Event Sourcing implementation was that, in order to replay and emulate past operations, the Events had to be closely matched against the code version that generated them since replaying an Event in a different code version could generate very different results. As such, every Event would carry the code version they were created in among their support information and, on every code release to production environment, the whole database was snapshotted so that no Events from the previous code version had to be replayed. In addition, whenever an old operation had to be evaluated, the source code would be rolled back to match the analyzed event version.

VI. Conclusion

Software programs usually suffer from lack of memory of its past states. The way many programs are developed, the internal state of its objects is constantly overwritten to better represent the current scenario. Despite this reality, many real life business scenarios could potentially benefit from the knowledge of a program's state evolution in time, such as Business Intelligence (BI) and production debugging. Event Sourcing presents itself as a different way to model Object Oriented software, in order to incorporate the concept of state evolution from its core. With its usage, it is expected that a program will be able to return to its past states and be replayed from them at any time, granting many benefits to analyzing past computations.

The adoption of Event Sourcing at Liera brought all the advertised advantages. In production debugging was made way easier. Once an issue was identified with a user, the debugging procedure was to simply get a production database mirror, identify the event triggered by the action reported by the user and erase the database from that point forward. After that, take the exact action the user took and study the outcome. This made complex production time problem solving a much faster operation allowing for higher standards of quality of service to be delivered. Past operation analysis was also greatly benefited from ES. Liera's parent company employs a complex logistics scheme with lots of validations and security procedures to enable very aggressive delivery deadlines pushing as far as thirty minutes to shipping after order confirmation. This scenario, which benefits from the production debugging capabilities of ES, also takes great advantage on past operation analysis. For instance, it is not a rare request on this scenario to evaluate why a particular item had its shopping denied several times during while fulfilling an order which eventually went overdue over a week ago. ES enables the team to recreate every attempt of including the item in the order, as it happened back then, and understand the reasons for each denial using the same process from production debugging. On this same scenario, ES also allows the team to understand the whole process involving an item denial by simply analyzing the recorded events which will show attempts from a supervisor to get clearance to enable shipping. This complex logistics model also took advantage of the BI-style report generating capabilities by transforming the events into new, business need specific models to answer questions about past pertains. A secondary, unpredicted, benefit of ES was that understanding the Event flow in the Event Store was simple enough even non-technical stakeholders were able to take part in analysis and discussion of many of the mentioned scenarios.

In the end, though, the expected benefits from Event Sourcing were achieved and the tradeoffs paid off but Event Sourcing must still be considered a niche solution while it matures to a less uncertain model.

Future works in the field could certainly explore benchmarking some of the decisions taken in this implementation such as the difference between MSSQL and MongoDB Event Sourcing, specifically regarding the performance loss on MSSQL due to BLOB usage and on MongoDB due to transaction support. Another interesting point to research and benchmark could be classic SQL-ORM based entity searching and materialization versus Event Sourcing approaches such as keywords and event replay or CQRS.

REFERENCES

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*, 3rd ed. Prentice Hall PTR, 2005.

[2] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch," in *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09. First International Conference on*, March 2009, pp. 36–43.

[3] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.

[4] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Prentice Hall, 2008.

[5] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 1st ed. Addison-Wesley Professional, 2003.

[6] K. G. Camargo, F. C. Ferrari, and S. C. Fabbri, "Characterising the state of the practice in software testing through a tmmi-based process," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, p. 7, 2015. [Online]. Available: http://dx.doi.org/10.1186/s40411-015-0019-9

[7] A. Santos, P. Alves, E. Figueiredo, and F. Ferrari, "Avoiding code pitfalls in aspect-oriented programming," *Science of Computer Programming*, vol. 119, pp. 31 – 50, 2016, selected papers of the Brazilian Symposium on Programming Languages 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642315004141

[8] I. Sommerville, *Software Engineering*, 10th ed. Pearson, 2015.

[9] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 34–43. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321639

[10] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 29–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.25

[11] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 4:1–4:28, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2110356.2110360

[12] S. Chaudhuri, U. Dayal, and V. Narasayya, "An overview of business intelligence technology," *Commun. ACM*, vol. 54, no. 8, pp. 88–98, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/1978542.1978562

[13] H. Watson and B. H. Wixom, "The current state of business intelligence," *Computer*, vol. 40, no. 9, pp. 96–99, Sept 2007.

[14] W. H. Inmon, "The data warehouse and data mining," *Commun. ACM*, vol. 39, no. 11, pp. 49–50, Nov. 1996. [Online]. Available: http://doi.acm.org/10.1145/240455.240470

[15] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, ser. Object Technology Series. Addison-Wesley, 1999.

[16] V. Vernon, *Implementing Domain-Driven Design*, 1st ed. Addison-Wesley Professional, 2013.

[17] B. Erb and F. Kargl, "Combining discrete event simulations and event sourcing," in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '14. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 51–55. [Online]. Available: https://doi.org/10.4108/icst.simutools.2014.254624

[18] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st ed. Addison-Wesley Professional, 2002.

[19] B. Erb, G. Habiger, and F. J. Hauck, "On the potential of event sourcing for retroactive actor-based programming," in *First Workshop on Programming Models and Languages for Distributed Computing*, ser. PMLDC '16. New York, NY, USA: ACM, 2016, pp. 4:1–4:5. [Online]. Available: http://doi.acm.org/10.1145/2957319.2957378

[20] B. Erb and F. Kargl, "A conceptual model for event-sourced graph computing," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. New York, NY, USA: ACM, 2015, pp. 352–355. [Online]. Available: http://doi.acm.org/10.1145/2675743.2776773

[21] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing nosql mongodb to an sql db," in *Proceedings of the 51st ACM Southeast Conference*, ser. ACMSE '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2498328.2500047

[22] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, Aug 2013, pp. 15–19.

[23] M. Stonebraker, "Sql databases v. nosql databases," *Commun. ACM*, vol. 53, no. 4, pp. 10–11, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721659