IMPROVING TRANSACTION ACCEPTANCE OF INCOHERENT UPDATES USING DYNAMIC MERGING IN A RELATIONAL DATABASE

A THESIS IN

Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment of
the requirements for the degree

MASTER OF SCIENCE

by
RYAN LINNEMAN

B.S., DeVry University-Kansas City, 2008
B.A., University of Missouri-Kansas City, 2011

Kansas City, Missouri
2015

IMPROVING TRANSACTION ACCEPTANCE OF INCOHERENT UPDATES USING DYNAMIC MERGING IN A RELATIONAL DATABASE

Ryan Linneman, Candidate for the Master of Science Degree

University of Missouri-Kansas City, 2015

ABSTRACT

Despite its tenure, mobile computing continues to move to the forefront of technology and business. This ever expansive field holds no shortages of opportunity for either party. Its benefits and demand are abundant but it is not without its challenges. Maintaining both data consistency and availability is one of the most challenging prospects for mobile computing. These difficulties are exacerbated by the unique ability of mobile platforms to disconnect for extended periods of time while continuing to function normally. Data collected and modified while in such a state poses considerable risk of abandon as there exists no static algorithm to determine that it is consistent when integrated back to the server.

This thesis proposes a mechanism to improve transaction acceptance without sacrificing consistency of the related data on both the client and server. Particular consideration is placed towards honoring data which a client may produce or modify while in a disconnected state. The underlying framework leverages merging strategies to resolve conflicts in data using a custom tiered dynamic merge granularity. The merge process is aided by a custom lock promotion scheme applied in the application layer at the server. The improved incoherence resolution process is then examined for impacts to the fate of such transactions and related bandwidth utilization.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis entitled “Improving Transaction Acceptance of Incoherent Updates Using Dynamic Merging In a Relational Database” presented by Ryan Linneman, candidate for the Master of Science degree, and certify that in their opinion, it is worthy of acceptance.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Caching data locally is a proven means of reducing run time dependencies in a system and enabling disconnected processing. As technology progresses, the complexity of cache implementation grows with it. Now that mobile computing is coming of age, it's plain to see how this holds true. Maintaining both data consistency and availability is no minor feat in the field of mobile computing and these opportunities continue to demand attention.

1.1 Motivation

Mobile computing fundamentally changes the landscape of cache design and management. It brings a smattering of new technical constraints into the fold such as limited client side computational capability, limited power availability, limited storage space, limited bandwidth, and the somewhat novel constraints of location dependencies, dynamic routing, and non-deterministic host availability [3, 4, 5, 6, 11, 12, 13, 17].

Significant effort has been invested in understanding these new constraints, how they relate to one another, and what can be done to mitigate any negative consequences each may have to the overall performance of a system. Some of these efforts will be reviewed in detail in the following chapter. Though, the net objective of these works consistently presents as one of reducing the occurrence of attempts to utilize historical, or stale, versions of data, be it for read or for write access. Furthermore, these works agree that an affinity for optimistic concurrency (OC) control mechanisms yields better performance than pessimistic concurrency (PC) alternatives, particularly for mobile environments.

The gains from OC measures are predominately yielded in lighter bandwidth consumption, and in turn, slower power depletion on the mobile unit. These gains result largely due to a reduced need to keep the radio tuner(s) of the device active. Considering the magnitude of the costs required to maintain a connection with a mobile unit, it is clear to see why stateless and OC models excel in mobile

```
UPDATE dbo.Employee
SET name = @p0
    ,modifiedOn = @now
WHERE employeeId = @p0
    AND modifiedOn = @p1
```

Listing 1.1: Optimistic update with incoherence dectection and prevention.

environments. Since an OC model late binds its ability to successfully commit a transaction, the need to keep real-time state of all data relevant to the transaction is bypassed. In addition, OC tends to support higher degrees of parallelism and result in fewer messages exchanged [14].

1.1.1 Optimistic Concurrency

OC works by applying time stamps, row version numbers, or a modified date-time field to records. Regardless of the field type, its value must be universally distinct over the life of the record and all updates must produce a new value for the field. Adhering to these constraints guarantees consistency in OC. Repeatition in the OC field for a particular record implies that record has returned to a previous state at a later point. This is at issue with any updates which occurred between the restoring version and the current version. Any version in that window is nondescript as it simultaneously describes both a historical value and a future value, which may never occur. Furthermore, if a client has a cached copy of the record which represents the restoring version, their cache has implicitly changed state from invalid to valid which could cause confusion. Updates against this model seeking to detect incoherence would look like that of listing 1.1.

This form of OC works by hiding the current version from the stale version when the stale version tries to update [7]. By simply looking at the number of rows affected by a given operation, it is easy to see that the record sought after is either removed or updated since last read.

1.1.2 Pessimistic Concurrency

PC works through applying a lock to items. Locks can be read, read-write, write, exclusive, shared, or any other variation the developer cares to code or is obliged to code per software requirements. Locks may be placed in-line or may be placed in a stand alone environment. A common approach is to place them in-line. Using the in-line approach, attributes are added to a tuple (fields to a table) which provide the functionality on the items to be locked with the id of the lock or holder. PC

```
UPDATE dbo.Employee
SET name = @p2
WHERE employeeId = @p0
    AND lockId = @p1
```

Listing 1.2: Pessimistic update honoring in line lock.

naturally works well for conflict prevention by eliminating it altogether. Common filter clauses take a form such as that used in listing 1.2.

1.1.3 Why Optimistic

Firstly, it should be understood that OC and PC in concurrency are not mutually exclusive across the scope of a solution and may even cooperate at an item level. For a particular subset of the entire problem domain of concurrency control, they are applied mutually exclusive. However, it is entirely feasible that one component of the concurrency management system may function in one mode while a sibling component may operate in the other as is the case in [6]. It is also possible to apply them in tandem on an individual item to reduce communication costs as will be shown.

As illustrated in Fig.1.1, the implications of each model share a relationship. As an implementation drifts into more PC, it becomes less available but also has fewer conflicts. While advancing in the OC spectrum yields higher availability but comes with the added cost of more conflicts. In the world of ubiquitous computing, the dominate motivating feature is availability. Point in case, this feature is implied by the name. When discussing mobile computing, it should be implied that where possible, availability assumes priority.

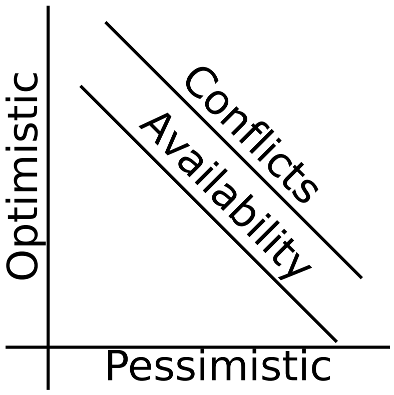


Fig. 1.1: How optimism and pessimism affect conflicts and availability of data.

This is not to say that availability is the end all however. If the data is untrustworthy then it has no meaning when it is available. Thus, the urge to make data available must be bridled with consistency. This directly correlates with a desire to move back towards the PC end in the figure to reduce potential conflicts. Balancing the requirements along these two traits exemplifies the need for robust concurrency control tailored to its application. The delicate balance in trade offs between availability and consistency paired with the similar parity of trade offs in mobile constraints serves to illuminate the scope of challenges addressed in the works cited.

In discussing mobile platforms, it is intuitive that OC garner favor as it produces higher levels availability over PC. The core challenges begin to surface when attempting to enforce consistency over these models at the client level in order to retain integrity and value of the data.

1.2 Problem Statement

These relationships appear to coalesce nicely to justify and simplify what goes into designing an efficient cache implementation for a mobile environment. However, caching in general suffers from an intrinsic flaw of incoherence.

PC suffers incoherence differently than OC. They both suffer this imperfection but PC is somewhat shielded from it. As illustrated in Fig. 1.2, after a transaction reads a value, another process may lock, update, and release it. Consequently, from the completion of t_3 to the completion of t_4, c_2 operates in an incoherent state. The impact is mitigated by the fact that in order to write, a lock must be acquired. Acquiring the lock frequently includes an implicit read [7] of the data as shown. Note how t_4 requests a $lock\, x$ and receives back x'. This effectively voids the incoherence and guarantees integrity of the write. Thus, c_2 is aware that it was incoherent but it does not matter since c_2 now has a lock with exclusive write access and has not yet written to x.

On the other hand, OC strategies prove particularly susceptible. The rather inane depiction in Fig.1.3 demonstrates the idiosyncratic relationship between OC schemes and incoherence. In this sequence, Once s accepts $write\, x'$ from c_1, c_2 is unconsciously operating with stale data.[1] The important take away from this trivial exercise is that the data read in t_1 can change at any time once a transaction completes. This means that if the transaction completes before streaming data back to the requester, as is frequently the case in environments like web services, the data may be stale by the time it is received by the requester. Thus, there really is no point in time at which c_2, or c_1 for that matter, can genuinely trust the data from s is authoritative until its next access. Here, incoherence results in implicitly fickle data in the local cache for mutable data. This flows as a direct result of the isolation tenant of ACID compliance for a traditional relational (RDBMS) or graph database, and of the inherent isolation witnessed in columnar, document-centric, and key-value stores. This also explains why the data of the OC environment for each $Client$ is intrinsically unreliable and why OC bears an elevated rate of transaction rollbacks over its PC counterpart.

[1] t_2 could just as easily come before t_1, or even concurrently with t_1 since they are both reading.

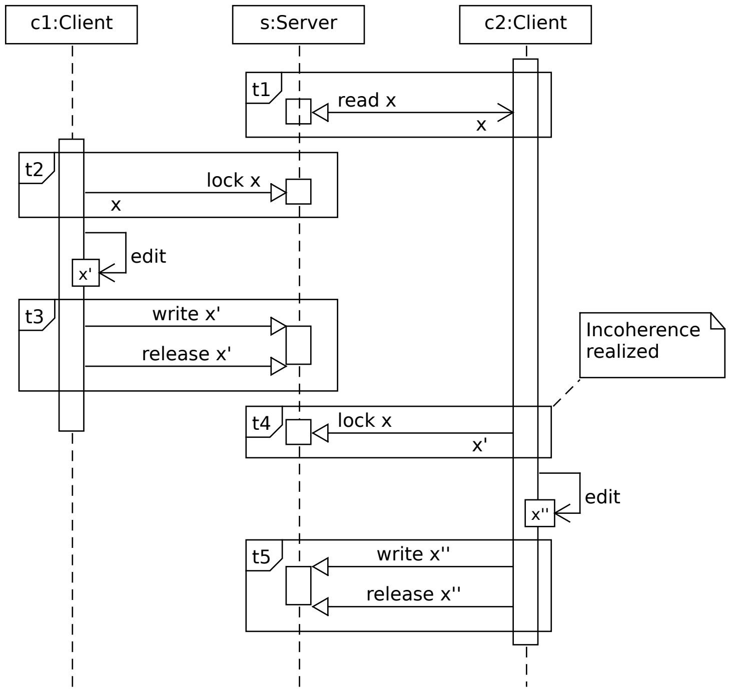


Fig. 1.2: Pessimistic ambivalence to incoherence

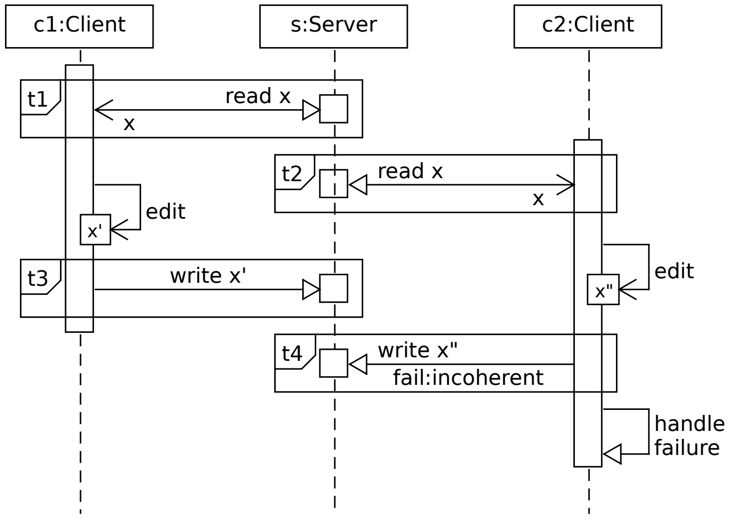


Fig. 1.3: Optimistic update incoherence

The volume of effort spent in mitigating the issues around weak consistency for read operations of OC models for mobile settings is entirely understandable. Given the ever growing prominence of mobile computing, usage of stale data poses significant potential for losses in productivity and expense. Furthermore, as demonstrated above, attempts to utilize stale data are an eventuality which must be accounted for. As such, the boilerplate response to an attempted update using incoherent data produces a transaction rollback. This behavior is of particular interest as the data may be temporal or location dependent and consequently costly or not possible to reacquire. This has obvious immediate costs but also buried costs in the form of influences on future schedules.

Though the attempted invalid update is guaranteed over time, PC locking is reserved for critical data points as the availability of data tends to trump its consistency. If transactions are rejected as a result of the OC scheme, the loss of the transactions represents an often acceptable[2], yet preventable, loss.

This leads to the premise of this thesis. Over time, rejections due to technical constraints imposed by the OC scheme will grow. Effort is made now to alleviate the accepted losses through enabling a higher level of acceptance by loosening the definition of incoherence within the application.

1.3 Outline

This thesis presents a solution to the preventable loss of optimistic caching models when bringing data back online without impacting the underlying caching architecture. Where the literature emphasizes prevention of stale writes, here the objective is to provide a more robust means of handling their unavoidable occurrence. Special consideration is given to the scenarios in which long term offline storage is sought to be reconciled, as this is near universally the area in which the caching models reviewed break down. The only two covered in the related works which may not entirely choke on such updates are covered in [1] and [15].

Leveraging shadow copy, OC to PC lock promotion, and record level data merging in a traditional relational database, it is shown that it is possible to improve the acceptance rate of transactions while maintaining coherence of each item, and in turn, consistency across the application as a whole.

Support for multiple merge strategies as well as variable levels of merge granularity are used to inflate the size of the reconcilable data set. A real world application serves as the basis for the proof

[2]While the loss is certainly undesirable, it is assumed to be acceptable. If the loss were unacceptable, the system should not have been built in such a way as to permit this behavior. In such a circumstance, this behavior would constitute a defect in the software in need corrective action.

of concept in implementation.

The effectiveness of dynamic merge granularity is vetted via tests cases. These tests are exercised using a variety of permutations of each merge strategy and granularity to provide perspective as to the performance implications of each model after a long term offline state as well as the improvements possible by exercising tiered dynamic merge granularity as a whole.

The rest of this thesis is organized as follows. Chapter 2 reviews related work in maintaining cache consistency in wireless networks and distributed systems. Chapter 3 calibrates the reader's perspective of an update request with the implications of accepting stale data. Then, the above approach to handle such request is expanded. Chapter 4 further expands upon the approach, offering an implementation to support it. Chapter 5 reviews findings from excising the work in the chapter 4. Finally, chapter 6 provides a brief synopsis of the work and concludes this thesis.

CHAPTER 2

RELATED WORK

A worthwhile level of effort has been invested in exploring the nuances to cache management in the mobile domain. This section explores inroads which have already been made.

2.1 Cache Concurrency

Cache concurrency is synonymous with cache invalidation and has been well researched in traditional environments. Such research has produced multiple variants of cache consistency including strong-, weak-, eventual-, delta-, probabilistic-, and probabilistic delta-consistency. Mobile computing has elevated the urgency to improve the schemes over the years. Each of the works presented here targets a distinct facet of the mobile conundrum with some overlap in between.

2.1.1 Invalidation Report

Invalidation Report (IR) was originally developed in [2] and has been cited, used, and extended extensively since its inception. This method of cache invalidation works by pushing more coarsely grained detail about updated information than had previously been exercised. Specifically, instead of invalidating a single item, the message to invalidate an item includes information about additional records which may be of interest to the intended recipient such as identifiers and timestamps of other items. This method has proven useful to so many, in part because consistency is assured to variable degrees and query and data availability is at most one IR transmission interval away. Furthermore, it mitigates cache maintenance protocol overhead by reducing the volume of request needed to assert validity.

IR supports synchronous and asynchronous flavors. The asynchronous mode is more akin to the legacy invalidation message as it is broadcast on update and is generally used to obtain some form of strong or delta consistency. In synchronous operation, the IR are batched for bulk delivery at a time to be determined later and clients block on cache misses due to invalidation until receiving the next IR. Synchronous mode is generally used to obtain some level of cache consistency less than strong consistency yet stronger than weak consistency.

2.1.2 Optimistic Concurrency Control with Update TimeStamp

The Optimistic Concurrency Control with Update TimeStamp (OCC-UTS) described in [11] is a hybrid approach which seeks to mitigate the up-link requirements by offsetting them with the broadcast capabilities of the mobile network. This is a stateless, backward validation protocol. A standard backward validation scheme evaluates the read sets of transactions to the write sets of other transactions [11, 8]. OCC-UTS is only a tweak to such validation so that rather than relying on inspection of a change set within each transaction and collisions between them, the serializability determining factor is the timestamp of the transaction. The logic for this protocol is split into two domains. One for server and the other for client.

As far as the client is concerned, its logic is not all that dissimilar from a conventional request based transaction manager. The client is weighted more heavily on the passive spectrum. When attempting to update, the client will issue a transaction to the server and wait for a yay or nay response back. Once received, if the result is in the affirmative, the client will commit the transaction locally and resume normal operation. If, however, the result is negative, the client will abort its local transaction and invalidate its cache. This guarantees that a subsequent attempt of similar nature will reacquire the data necessary to facilitate the transaction, thus increasing the likelihood that it will succeed.

The server balances the client's passive behavior by giving more credence to the active end of the spectrum. The server logic emphasizes creating the IR. The IR is constructed by examining the timestamp of update transactions and the updated items. If the timestamp for the update of the data item falls within the update window, then it is added to the potential IR queue. If there exists any active transaction referencing the data item then the record is added to the pending IR. When the window elapses, the IR is broadcast rather than issued to specific hosts. In this way, majority of the heavy lifting for transaction processing is offloaded from the server to the clients performing the updates.

This protocol is robust enough to function independent of cache availability on each local client. This means that the protocol supports a heterogeneous cache availability environment [11]. The primary difference resides in the client processing in that it can bypass all cache references without changing its execution model. Regardless of the mode of operation of the client, the server's execution remains immutable.

2.1.3 Probability-Based Callback

Where OCC-UTS emphasizes minimizing communication, Probability-Based Callback (pCB) targets communication and query latency with added value placed on its ability to adapt to heterogeneous networks prominent in mobile networks [16]. This protocol is predicated on the notion that the communication cost and the access delay are adjustable and that they may be adjusted to fall within accepted values of the standard Callback (CB) and Push schemes. The valuable ability to make this claim comes at the significant cost of converting the server behavior on update to become non-deterministic. Consequently, this scheme is not independently strongly consistent. Strong consistency under this scheme requires implementing locks at the application layer to prevent dirty reads and inconsistent updates [16]. The authors intentionally neglected this aspect, making the assumption that this behavior is likely to be implemented in the application layer any way.

Another important assumption made by this scheme is that the communication channel bandwidths in mobile networks are asynchronous, and that the down-link has greater bandwidth than the up-link. This protocol will evaluate the communication costs of each up and down link control message, the down link data transmission, and the latency cost for control and data messages independently. The probability that an item becomes marked as invalid is exponentially weighted by its data communication cost. The cheaper it cost, the more likely it is to be flagged as invalid after an update. When an item is marked as invalid, the probability that it will be toggled back to valid is exponentially tied to the control costs on both the down and up-link channels. Thus the more expensive the control costs become for the item, the more likely it is to remain marked invalid so that when the IR is issued, the clients using it need fewer control messages to keep it in sync. This model is very effective in mitigating costs associated with transactions in mobile space and can easily self adjust to workload as the day progresses and load shifts when people move. One of the more difficult things to adjust to is the indeterminate nature of the IR which are issued. It is clear that in order to retain strong cache consistency, more must be done to resolve the indeterminacy in the update procedure or risk lost updates and other artifacts of lack of governance.

2.1.4 Adaptive Energy Efficient Cache Invalidation Scheme

Adaptive Energy Efficient Cache Invalidation Scheme (AEECIS) is a newer approach discussed in [12]. This scheme is adaptive to current time constraints, volume of clients requesting updated data, sensitive to bandwidth consumption and connection counts with the primary objective being energy efficiency in the mobile unit (MU) facilitated by optimizing the time it spends with the tuner active. It is important to note that this is a stateful scheme since it is conscious of the volume of users requesting data as well as the data items being requested.

In this scheme, the server operates in one of three modes: slow, fast, and super-fast. While in slow mode, the server IR being broadcast are limited to coarsely grained group identifiers. This is a consequence of low request volume. Since the server can quite clearly tell that there are not many incoming requests for updated data, it is more efficient to transmit everything in larger blocks so that the consumer may make fewer requests and the channel is not congested with an excess of similar data for others requesting the same data.

At some point, the request volume will pick up within a specific time interval. At such time, the server will promote to fast mode. When it makes this step, the grain of the IR is refined, but is still only transmitting invalidation identifiers. This allows clients to retain their cache validation and invoke requests for only the select items which have decayed. Finally, if the server has detected a significant hike in the request volume, it will jump up to super-fast mode. In this mode, the server will be transmitting complete objects as opposed to just their identifiers. This helps to allow the large scale usage of a specific data item while eliminating the need for the MU to individually request it. Eventually, the workload will subside and begin to decay again. As this occurs, the server will begin stepping backward, from super-fast to fast mode, then landing back to slow. This scheme was shown to be very effective in stably being the most efficient when pitted against widely used schemes [12].

2.1.5 Distributed Cache Invalidation Mechanism

In [6], a means of maintaining cache consistency in mobile ad hoc networks (MANETs) named Distributed Cache Invalidation Mechanism (DCIM) is presented. This is described as a client adaptive time-to-live (TTL) based pull scheme, similar to the natural client server architecture which dominates the internet today. Its objective is to statelessly improve the efficiency of cache updates within. Inside the MANET the model architecture resembles that of a content delivery network (CDN) and the concept is actually quite similar. Moving the data closer to the requesting nodes reduces the volume of hops to acquire the data and in turn the latency to satisfy a query. Having the data available within the MANET has the added advantage of each client node not needing to establish a connection back to the server. Another benefit and distinction from CDN's is that every node within the MANET may provide content as a content node (CN). Again, significantly reducing the necessary number of hops for peers to acquire data.

Moreover, DCIM introduces a custom cache invalidation procedure to ensure content within the MANET is up to date. In DCIM, CN's opportunistically use time-to-live (TTL) values to derive update patterns and predict invalidation schedules for data which they are indexing. When communicating with the server to update local content, the CN also injects administration messages for the server which significantly reduces the requirements of the CN for connecting to the source server.

2.1.6 Cooperative Approach to Cache Consistency

This method, presented in [18], is a hybrid push-pull IR based cache consistency approach designed for wireless mesh networks (WMN). In this approach there are four classes of entity; server, gateway, router, and client. Servers are stateful, retaining a list of networks, not clients, which are caching specific items. Upon update, the server issues IR to the the gateway(s) of relevant WMNs predicated on its own cache placement model. Within a WMN, the gateway(s) own all IR and buffer them to avoid network congestion. The IR are periodically issued from the gateway through the routers, also known as MAPs, determined by elapsed time since last issue or optimal network packet size. When issued, the MAPs process and forward the IR to all clients directly connected. Upon receipt of an IR, clients and MAPs process the IR, invalidating its local cache as necessary. Since clients may be offline, they may miss IR. If a client misses an IR, it can request a resend. IR which are re-issued may come from the gateway or the client's MAP. When any node receives a request for an IR, if the IR is known to the node, the node will send it. However, if the node does not have the requested IR, it will discard the request if another request for the same IR is pending at that node. When the IR requested has not been requested previously, the request will be placed in a pending queue and forwarded up stream.

2.1.7 Extended LRU

Extended LRU (E-LRU) [9] is an interesting and simple yet effective revision to the time tested least recently used (LRU) cache eviction policy. As the author points out, LRU is a simple scheme but often ineffective because it does not consider enough inputs and because other studies cited in that work have shown object cache volume to perform object cache size consistently.

This technique partitions the cache into two tiers before applying the LRU principle. First Items which have experienced only one request are given greater priority for eviction over items with more than one. These items are treated with LRU directly. Next, items with more than one reference are inspected for the inter arrival time of their two most recent accesses. Whatever item has the greatest inter arrival time is elected for eviction.

Furthermore, as object cache size impacts cache performance, only items of size no greater than half the total cache size are permitted entry.

2.1.8 Fast Wireless Data Access Scheme

Fast Wireless Data Access Scheme (FW-DAS) [10] is a revised pairing of Poll Each Read (PER) and CallBack (CB). In this method, the network is bisected into the last hop between the mobile unit (MU) and the base station (BS) and the wired route between BS and the application server (AS). Caches are created on each MU and a larger cache exists on the BS.

Within the mobile space, the MU cache uses PER to invalidate its local data. This ensures that MU is spending a minimal amount of time tuned into the BS for cache maintenance. Alternatively, the BS is using CB to ensure data is available with minimal latency.

In this setup, the AS is stateful so as to know what BS to notify when updates to an object occur. The BS is at least semi-stateful as both the BS and AS keep tabs on the popularity of items. Knowing the popularity of an item serves to improve latency by allowing both nodes to be proactive in deciding if a data item should be included as a part of the standard control messages in use.

2.1.9 Eventual consistency

Eventual consistency [15] is a form of weak consistency which guarantees that in the absence of changes to data, all nodes participating will eventually hold the same state. The point in time at which such a state could be achieved is non-deterministic and in practice does not happen at a system level in large scale deployments. Individual data items however, will certainly become consistent over time as the delay between their writes grows.

Limiting the system with this constraint allows for the refinement of cache granularity and offers a few variations in other rules which may be applied to obtain styles of consistency conducive to specific applications. Specifically, eventual consistency comments on the notions of causal-, read-your-writes-, session-, monotonic read-, and monotonic write-consistency. This work also points out that this guarantee does not preclude combining the logic used in these modes. Furthermore, these modes only outline rules governing how queries are to be processed, not the behavior or changes in the constraints of the system.

2.2 Interpretation

Thus far, emphasis has been placed on the myriad of cache concurrency forms available. These mechanisms for cache management serve to highlight the varying opportunities in distributed data. As this thesis intends to improve the acceptance rate of transactions executed under erroneously optimistic settings, focus will now be turned from cache strategies to evaluate other relevant propositions in this area.

2.2.1 Challenging Conventional Notions of Isolation and Consistency

Till now, the literature has worked toward achieving a consistent state amongst distributed nodes via some form of communicating shared state between all relevant parties. This communication results in costly overhead and, as a matter of consequence, markedly reduced concurrency in standard modes of operation. Invariant confluence [1] (I-confluence) elects to outright side step this overhead where possible. The insight here is that conventional database isolation mechanisms operate on a strictly read/write operation evaluation and a more macro view which considers the requirements and behaviors of a system is preferred. Using this approach it is possible to materially reduce the overhead costs incurred by coordination between nodes and between operations, often eliminating it.

I-confluence asserts that “coordination can only be avoided if all local commit decisions are globally valid. [1]” Where it occurs that commit decisions are not globally valid, coordination must be exercised to determine the proper course of action. At the time of this writing, there does not yet commercially exist any query plan estimator capable of evaluating the satisfiability on a per query basis in real time. However, using static analysis of a relational database’s schema and knowledge of the invariants of the software system at large, I-confluence was shown to be viable, practical, minimally sufficient to ensure isolation, and capable of satisfying the needs of concurrency control while all but obviating it.

CHAPTER 3

APPROACH

Many issues in caching arise from the concern of the usage of stale data. The existing works' emphasis is on preventing stale cache. The effectiveness of all of them deteriorate directly proportionally to the duration for which a mobile unit and its cached data are offline. This thesis approaches the issue of stale writes from an alternative angle and moves to resolve conflicting updates. In the same way which PC works to preempt conflicts and OC moves to resolve them, the approach here is to resolve the occurrence when it does surface.

3.1 ACID Compliance

If a transaction is to have any value it must be ACID (Atomicity, Consistency, Isolation, and Durability) compliant, or at least observe the properties of ACID as in the context of [1]. This compliance is what gives transactions the ability to provide the reliability and predictability required by businesses. Achieving this compliance can be challenging enough in its own right, however introduce the need to support intermittent connectivity and a very challenging problem is presented. Specifically, how does one continue to assert ACID compliance in a system while not only its data, but also its operational context is completely isolated (offline)? This question is relevant despite knowledge or predictability of the occurrence or duration of such isolation.

Remaining functional at all times is the objective of mobility and caching is the tool to enable it. However, as previously demonstrated, no data in cache may be considered as trustworthy so long as it is externally mutable. Clearly this has ramifications to the ACID properties of business or application level transactions which operate beyond the bounds of raw system level transactions as illustrated in figures 1.2 and 1.3. To better understand the approach, it's best to have a clear understanding of what those implications are.

State	Description
Current	Values on an item accurately reflect the presently persisted values in the database.
Stale	Values on an item do not reflect the persisted state in the database but do reflect values which were persisted at a earlier point in time.
DNE	Values on an item do not reflect the persisted values in the database and do not reflect values persisted at a earlier point in time.

Table 3.1: Proper States of a Data Item

3.1.1 Durability

For online systems, durability is a trivial responsibility to manage. The fact that the DB guarantees durability and is also the source of truth eliminates durability as a concern. For offline systems, a similar situation ensues. The primary difference between the two being that the local persistence environment does not necessarily guarantee durability. If it does not, the developer(s) must provide the durability in the application when interacting with the persistence environment.

Durability varies and frequently must be considered across both local and origin persistence environments for intermittently connected systems. In practice, it is common to disregard offline durability constraints and only worry about durability when the application restores online status. This stems from the notion that in spite of a client moving to an offline state frequently, it will also be online frequently as well. It is assumed that work lost (if any) is of minimal regard due to the regularity of restoring online state. This behavior coupled with the mobile constraints outlined in the introduction often justify passive attention to the mobile persistence layer.

3.1.2 Atomicity

By definition, the action or collection of actions taken either complete in whole or never happened. Atomicity is frequently and easily relegated up the software environment stack to the database (DB). However, when discussing isolation, it will be demonstrated that where concurrency exists it must be managed otherwise there can be no atomicity. The reason for this is one of dirty reads and hidden updates and a disconnect of the in-memory or locally cached objects (CO) from the DB state. An object read from the DB through a transaction will represent one of those presented in table 3.1.

When the CO is initialized from the DB it represents the current state. Once modified locally, it represents the DNE (Does Not Exist) state. Once the modified version is saved it represents the current state again. If other instances exist then they represent the stale state until such time as they are reinitialized from the DB. If an attempt to update the DB with a modified version fails, then the CO remains in the DNE state. Note that there is no future state. This is an artifact of atomicity as the state either exists or it does not in the lineage of the data item represented by CO. The future state may be resolved as such only after it has become the current state. It's useful to understand this in the context of mobile and distributed computing as there are many possible concurrent futures unfolding at any given time for a data item yet only one may constitute the next state of the the object. Thus, if an update fails, at no point in its history has the data item ever held the state represented by CO, as such it does not exist.

3.1.3 Consistency

Consistency is what ACID is all about. All other properties make consistency possible and consistency is the collective objective of the ACID properties of a DB. Another way to phrase consistency is precision. Regardless of the accuracy of data in the DB, it must be precise, never wavering. It is up to the business rules of the system and its users to determine the accuracy of the data. However, regardless of how faulty the data is, it must always give the same values when requested. The DB is a tool of precision, the applications built around the DB are tools of accuracy. They facilitate the business rules and user experience around managing data within the DB but the DB always remains the steward of the data and must always present it consistently.

The motivation behind consistency is source of truth. In standard DB models, this is easy to assert and understand because the DB is the source of truth. However, in an intermittently connected or distributed environment any one client may have multiple sources of truth. Identifying which is the legitimate source of truth can be tricky and may fluctuate with business rules. As a result, source of truth becomes subjective and implementation specific. In a mobile device, the source of truth may be one of

- local cache
- resolved through Location Dependent Query (LDQ)
- a static DB requiring connection for re-query of the truth at each request

In any case, the question of which one represents the latest and most legitimate truth is always significant. While online, the answer is trivial. Simply return to the origin of the data and re-request. The fact that the data has already been requested from that origin implies a trust from the client to the origin that the client believes the origin to be the source of truth. While offline, the answer is also simple, there is either none or the local cache is the source of truth. Assuming there is a local cache and it is mutable while the environment is offline, when the application comes back online after the cache has been modified, should it be considered the source of truth or the actual origin? What if the origin record has been updated by another consumer while offline, causing the local cache to diverge from the origin? Optimistically using a version or last modified timestamp is commonly used to answer these questions and resolve the latest and greatest. If content is cached locally, the system can continue to work just as if they were online, the draw back is that it may lose work due to irreconcilable conflicts when the system comes back online due to this divergence. Of course this can be remedied with concurrency management which brings up the topic of Isolation.

3.1.4 Isolation

Where concurrency exists isolation must be evaluated. If isolation does not exist, atomicity and consistency can not be guaranteed. Concurrency is implicit when discussing isolation because in the absence of concurrency everything is isolated. In practice, there are two ways to address concurrency conflicts; prevention and resolution.

Prevention is very basic and a simple approach to understand. In PC, indeterminacy is prevented by disallowing concurrent edits. Since concurrency is removed from the system, everything becomes isolated. The problem with prevention is obvious, particularly for mobile platforms, as the availability of data is bottlenecked. If there is no concurrency, the bandwidth of the system is materially impeded and resource utilization is significantly reduced. The term pessimistic is used to describe this behavior as it stems from the assumption that if one does not proactively reserve its intent to have exclusive access, then it will be unable to obtain that access later when it becomes necessary.

Alternatively, if a resolution approach is taken, the point in time at which the CO is saved is when control measures kick into effect. A resolution model must detect conflicting changes to a record which has been modified since read. As a result, conflict resolution is where things can become very obfuscated. There are many common patterns to detect and resolve such conflicts to achieve concurrent edits. Conflict resolution methods are lumped together under the umbrella term optimistic.

A common and easily understood example for developers is in source control management systems. Multiple developers frequently work on the same files and must merge changes to ensure integrity of the source files. The same behavior applies and is sought here. Depending on the business rules of the system, this may be automated or require user intervention. In either case the record would reflect both changes because the data was disjoint or only the immediate predecessor's written values because the data manipulated was irreconcilable. In addition to the business rules, it is also important to consider that technical and performance constraints of the system have tremendous influence on the concurrency model to use.

Since pessimistic locking is all about conflict prevention, it is best reserved for scenarios where the expectation or risk of conflict and rollback are high. Conversely, optimistic locking is well suited for high concurrency, low risk updates. Low risk being the risk of conflict or the cost of loosing the data being low. Excellent scenarios for this model include read-optimized environments or well partitioned data. The partitioning clause here is what makes todays NoSQL stores so useful for distributed environments. For a mobile platform or other platform which is intermittently connected another approach is required.

3.2 Applying ACID to Naively-Optimistic Transactions

A transaction in the DB begins only after a connection to the DB is established. However, it is commonly understood that best practice is to keep transactions as short lived as possible to limit resource contention. To facilitate this, the CO is modified prior to connecting to the DB. Therefore, the only time the DB is in use is during the connect, update, and commit/abort phases of the object life-cycle. Deductively, the CO is modified before the DB transaction begins.

With respect to the sequence of events leading to an update on a given data item, it could be argued that the transaction begins when the object is initialized from the DB or when it begins to be modified. For the sake of system integrity, it is more practical to assume the former marks the beginning of the transaction. To accept the latter implies that the data being written was not read within the same transaction which creates an opportunity for conflicting updates. However, in practice, this distinction is semantically null for OC.

3.2.1 Constraining Facets

To assume reading an object from the DB marks the beginning of the application transaction T_A simplifies the thinking about the problem of its concurrency. To adopt this position allows that T_A begins as a parent transaction to all subsequent DB transactions (T_D) in its domain, including the initial read to the final write of a given data item. In this way, despite the presence or absence of failures to commit, $T_A \coloneqq \{T_{D1}, T_{D2}, ..., T_{Dn}\}$ remains an isolated, atomic unit. This gives rise to the first invariant constraint and assumption made in this solution. For every T_D under the supervision of T_A, it is always true that the success of T_D implies the success of T_A. More formally $\Box\,(T_D \implies T_A)$ where $T_D \in T_A$. In addition, any transaction T_X operating in the system must operate under the supervision of some T_A and therefore be a T_D. At this time, it is worth pointing out that T_A is conceptual in nature, thus, it is possible that $T_D \equiv T_A$.

Violations of this relationship result in incoherence anomalies as multiple T_D witness intermediate state of T_A for which $T_D \notin T_A$ when they transgress their isolation boundaries. However, T_A represents a potentially long running process and it may require updated data to complete its target objective. As such, T_A is viewed similarly to the states of a data item outlined in table 3.1. To elaborate, T_A may be viewed more akin a method of check pointing in that, if a T_{Di+1} fails, the state of T_{Di} remains the current state of T_A. This also ensures that at any given time, if T_A should fail to complete after a T_{Dj} has completed, the relationship $T_D \implies T_A$ holds from the perspective of the DB. Consequently, to the DB, all T_A are always in a completed state, eliminating contention between them as their observable state is always consistent. This has the added benefit that T_A is technically permitted to never end or may even fail (which would violate $T_D \implies T_A$) as from the outside looking in, view of T_A is always completed.

Figure 3.1 graphically represents this concept. The dithered T_A represent previously publicly visible state. The dotted lines are paths which would have been taken had the underlying T_{Di} been successful in committing. The dotted line from T_A to T_{Di+1} shows that T_{Di+1} was the next state of T_A but T_A rolled back to its last known consistent state of T_{Di} upon the failing of T_{Di+1}.

3.2.2 Leveraging Hybrid Locking

Applying these observations in a real world example is complex but yields a solution to allow for maximum concurrency where the bounds of conflict prevention cannot be known or enforced in advance. As previously noted, OC and PC locking should not be considered as mutually exclusive and the following illustrates why.

A hybrid model of OC and PC works well to bridge and garner the strengths of each. In this hybrid model, both version and locking information is used. If applying the locks in line as was done back in listing 1.2, a simple update would like the one in listing 3.1.

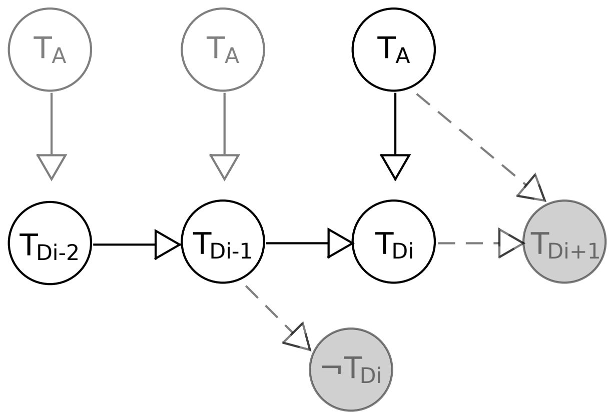


Fig. 3.1: T_A Lifetime Events

```
UPDATE dbo.Employee
SET Name = @p0
WHERE EmployeeId = @p1
    AND version = @version     -- I have optimistic lock
    AND (lock_id IS NULL       -- noone holds exclusive access
        OR lock_id = @lock_id) -- I hold exclusive access
```

Listing 3.1: An optimistic-pessimistic hybrid update query.

Through this query model, the optimistic model is ignorant of anything special with another potentially competing lock type. An observed behavior to using this model is that OC locks may be promoted to PC but the inverse does not hold. One way to consider this is any process which reads a $dbo.Employee$ implicitly obtained an OC lock since the version is known and current. The acts of applying and releasing PC locks from the record involve writes to the record which in turn provoke new version identities. Simply by acquiring a PC lock, all outstanding OC locks are cleared. Any new consumers which read the record now read it with the pessimistic lock applied and now give precedence to the PC form.

A PC lock may not be demoted because all implicitly obtained OC locks were invalidated through its acquisition. To demote to the OC form implies the version is rolled back, which effectively restores all previously cleared OC locks and violates the assumption of OC that a version identity is unique in time. Also, demotion would mean that the version generated when a PC lock was obtained never existed. Consequently, any clients consuming that version (at least one, the one which acquired it) are consuming an inconsistent, non durable state of the object from a system which should be ACID compliant.

One significant advantage this hybrid model offers over either one independently is particularly useful in mobile and distributed settings. Specifically, it allows for greater concurrency of data by using the normal OC mechanism most of the time. However, when greater governance is necessary, a data item under such control may have access throttled back to a serial model. In a mobile scenario where access is being made via the costly cell channels, the requests to acquire and release the PC lock are obviated unless absolutely necessary.

In the end, in order to support transactions in an environment which experiences intermittent connectivity intentionally, locks may be leveraged to provide isolation which in turn allows the host system to satisfy ACID compliance. However, locking alone does not provide ACID compliance and each property is impacted by a loss in connectivity. Thus, achieving ACID compliance requires through planning prior to implementation within a mobile platform.

3.3 Negotiating State After Divergence

Once connectivity is re-established yet before the application is fully brought online, it is necessary to perform a concurrency check with the origin. This is similar in concept to the act of bringing a DB online in that the DB must process its logs to restore its last consistent state before it terminated. Since offline changes result in two sources of truth for a given data item, the client and server must negotiate to reach a consensus as to what constitutes the actual current state of the data item. Table 3.4 at the end of this section provides quick reference in interpreting the negotiation process described below. Figure 3.2 offers a cursory high level overview from the client.

3.3.1 Client Side

In order to establish the terms of negotiation preparations must be made. This step starts before the client moves into an offline state and is not particularly interesting in the scope of negotiation but is crucial to ensure the client remains efficacious while offline. Simply put, data must be cached locally so as to be made available while offline. Furthermore, as the client will be offline, a rather inflated dataset is needed to be available for the client to operate on. This implies a form of partitioning is necessary. The online partition of data cached will be denoted as p. Locally the partitioning scheme used to produce p is irrelevant.

Once necessary data is cached, the client is able to detach from the network and roam independently. At this point, the cache is logically partitioned into two regions. A source of truth segment,

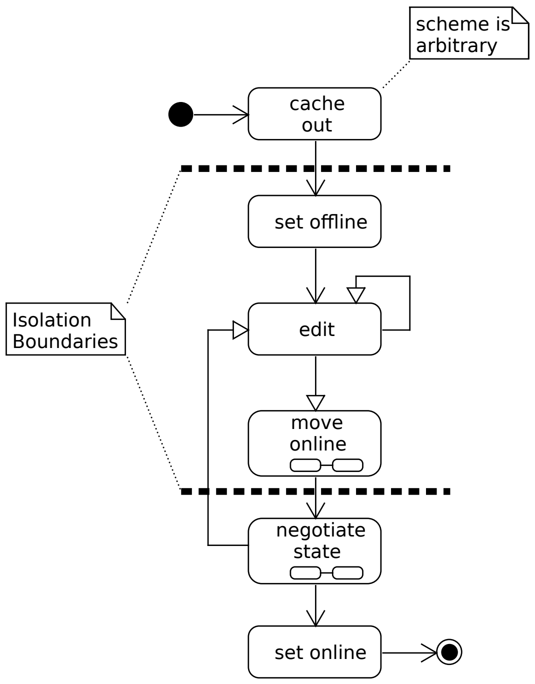


Fig. 3.2: High Level Overview of Process

denoted as $\top$, and a sandbox area denoted as $\perp$. While in the offline state, any item modified by the client is shadow copied from $\top$ into $\perp$ for manipulation. In this way, $\top$ emulates the DB from an online state and $\perp$ assumes the responsibility of local cache to the application. These views of responsibility are somewhat superficial however. Independently, each is not particularly useful. Combined, they are more appropriately viewed as the inner workings of a transaction which executes within the DB. For the remainder of the client's offline state $\perp$ will be leveraged as the source of truth for data items which it retains. Beyond the local cache management, the application's offline functionality is indistinct from the online state.

Partitioning the cache primed the terms of the state negotiation to follow. In the client domain, attempting to return to an online state does not depart much from its standard operation either. The main difference is in the steps taken prior to denoting that the client is indeed online.

When restoring connectivity, the client iterates over all items recorded in $\perp$. If $\perp$ is empty, then there is no work to restore and the client simply invalidates its cache[1] and marks itself as online. However, assuming $\perp \neq \emptyset$, each item in $\perp$ is paired with its source from $\top$ and appended as a tuple with its corresponding action to the change set $\Delta \coloneqq \{D_1, D_2, ..., D_n\}$; where $D \coloneqq \{d_1, d_2, ..., d_n\}$ and

[1]This could be satisfied by one of many options such as those from the previous chapter. As the interest here is in a partition of data, it is assumed that the volume of data is large enough to discredit the validity of anything more than simple invalidation.

d_i represents a discrete field from data item D. Items which are new are marked as such and have no corresponding $\top$ value. Deletes are complementary, having no $\perp$ within Δ. Items in $\top$ which were not in $\perp$ just wait in the cache for further direction. Compiling the change set is a basic join operation and simply results in a scalar message defined as $m := \{p, s, b, \Delta\}$ where s signifies the state of negotiation and b is to be defined later. In this initial message $s = initial$. This message may then be relayed to the server for processing as a single request, minimizing the overhead of communication costs.

The server processes the change set and yields a result back to the client. The result returned is also of type m and identified as m'. For responses, m' specifies the server's state of the client's request, where $s' \in \{conflicting, accepted, error\}$ and b remains to be defined.

The properties of Δ' differ from the version sent by the client. In the response, Δ' consists of update, new, and delete instructions, the same as Δ except all records in Δ' reflect the server's current state of the items submitted in Δ. Receiving Δ' is outlined as follows.

For each D' in Δ' if $D' \in \Delta'_{delete}$, remove D from $\top$. If $D' \in \Delta'_{insert}$, add D' to $\top$. If $D' \in \Delta'_{update}$, update $D^{\top}$ to reflect D' and add D' to a collection to be reconciled later. Since all items D in Δ leverage $\perp$ as their individual source of truth, isolation is preserved on round trips during negotiation. Now that $\top$ reflects the server's known state for the items in question, $\top$ may be used for comparison to reconcile outstanding conflicts which the server was unable to handle. At this point, $\top$ reflects a mingling of object state for items read when p was cached and the most current known state of items in p for which the user is attempting to modify. $\perp$ has seen no changes.

Armed with comprehensive knowledge of the data in p, the user may reconcile conflicts beyond the domain of the system to automatically resolve. Through the act for reconciling, $\perp$ will be modified to reflect the user's intent in light of changes made remotely. Figure 3.3 provides a visual formalization of the process described above.

3.3.2 Server Side

In this approach, the server is responsible to persist or advise the client, but not both. There is one exception to this which involves governance of isolation and the hybrid approach to locking outlined above. In select cases, it may be desirable to throttle back access to data while negotiating state changes for extended offline durations. Specifically, if the data being updated sees frequent updates and the incoming change set is critical or if the the data submitted from the client is sufficiently large, it may prove advantageous to shut down concurrency to a serial model. In such settings, the p itself may

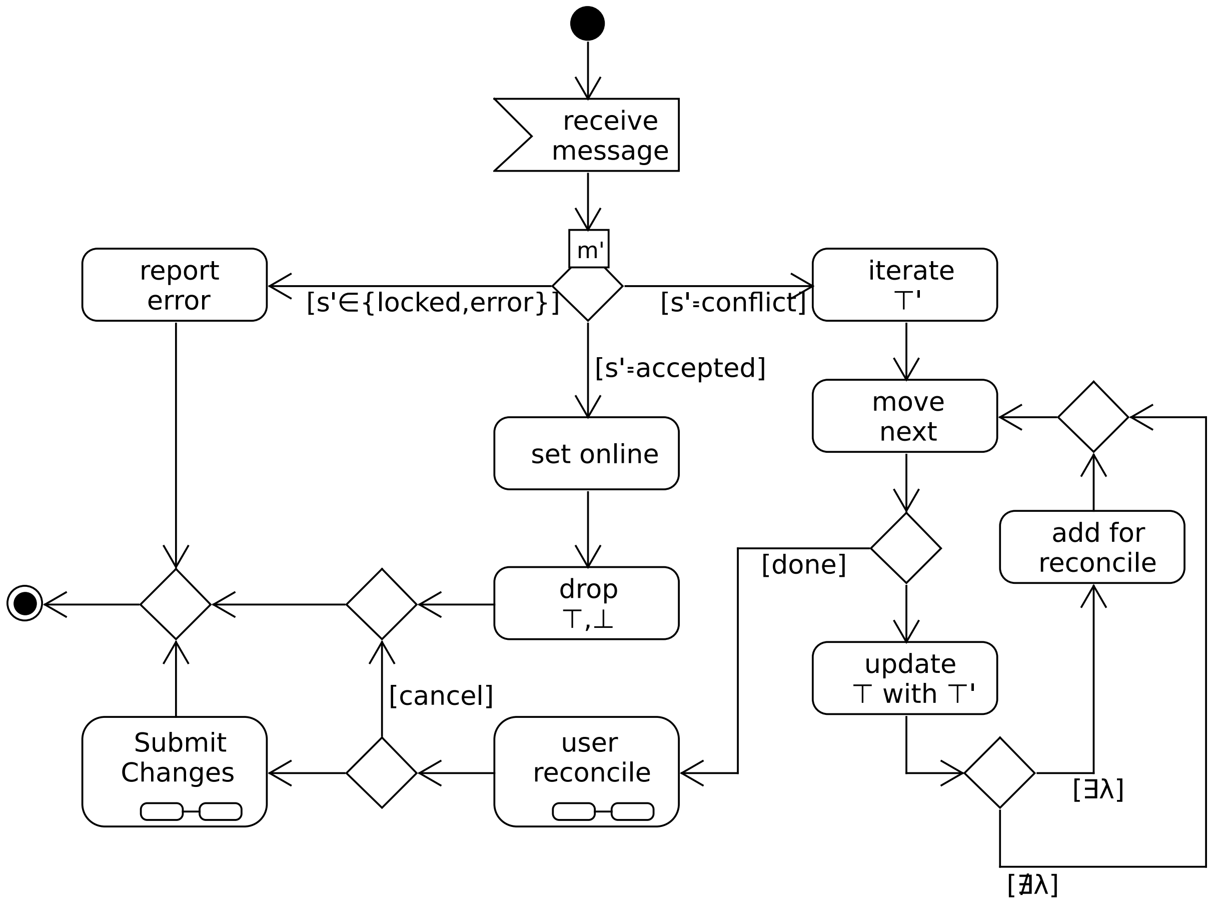


Fig. 3.3: Client Negotiation Cycle

Direction	$b = true$	$b = false$	$b = discretionary$
Client to Server	Client desires PC lock	Client desires release PC lock	Server decides acquisition
Server to Client	Client holds PC lock	PC lock exists but not held by client	No PC lock exists

Table 3.2: Lookup of b meaning

be PC locked to eliminate conflicting updates as a potential source of failure for future requests. This is where the b value of m comes into play.

For a given message m defined above, b indicates a desire to effect change in the PC structure of p and is defined as $b \in \{discretionary, true, false\}$. See table 3.2 for detailed explanation of interpretation of these values. Violation of the server's persistence clause may occur if $b = true$ and there is no PC lock on p, or when $b = false$ and the requester holds a PC lock on p. In the event that $b = discretionary$ the server shall elect a path reflecting $b \in \{true, false\}$ determined by some variable factor such as the size of the message.

When a message m arrives at the server, the server first inspects b. If $b = discretionary$ and the message size breaches some threshold, $|m| \geq \epsilon$, the server will assume $b = true$. If $b = true$, a PC lock is attempted. If a lock cannot be acquired, either due to error or another process holds such a lock on p, the request is terminated, returning $s' = error$, including details about the cause and setting b' appropriately. If no lock is to be granted, the system checks if another process holds a PC lock on p. If one is found and the lock holder does not match the requester, the request is terminated as above with $b' = false$. If the requester holds the lock, $b' = true$ and if no PC lock has been applied to p then $b' = discretionary$.

Once preliminaries are out of the way, the server moves on to reconciliation. Reconciliation is performed using a three way merge between the original value determined by $d^{\top}$, the requested value from $d^{\perp}$ and the current value of the field from the record in the DB identified as d'.

The merge is fairly basic but may be improved through the framework provided in the next chapter on implementation. For context, and example implementation is described here. During merge, all manipulation is done on $D'^{\perp}$. $D'^{\perp}$is seeded as $D'^{\top}$ then has changes from $D^{\perp}$ played atop it. $D'^{\perp}$ may be $D^{\perp}$, however, in the interest of performance, $D'^{\perp}$ could be an instance of D' distinct from D'. If a conflict is found, the change set will be rejected. If $D'^{\perp} \neq D'$ then dropping the changes simply requires updating pointers and does not involve any database activity. When a change set is rejected,

State	Implication	Rule	Conflict
Current	noop	$d^{\perp} = d'^{\top}$	no
Stale	n/a	$d^{\top} \neq d'^{\top}$	depends on $d^{\perp}$ and $d'^{\top}$
DNE	next value	$d^{\top} = d'^{\top}$ and $d^{\perp} \neq d'^{\top}$	no
DNE	divergent	$d^{\top} \neq d'^{\top}$and $d^{\perp} \neq d'^{\top}$	yes

Table 3.3: Formal object state definitions

the current DB values are fed back to the client. If$D'^{\perp} \not\equiv D'$ then $D'^{\perp}$should be pre-seeded from D' so that values absent from $D^{\perp}$ which exist in $D'^{\top}$ carry forward. This allows for optimizations in the data sent by the client by eliminating the need to send unmodified data up stream.[2]

Prior to invoking the custom framework, a remedial field by field evaluation is done. After all $d^{\perp}$ are cloned onto $d'^{\perp}$ they are evaluated in the following manner. If the new value does not match the current ($d'^{\perp} \neq d'^{\top}$) then the current value is compared to the original ($d'^{\top} = d^{\top}$). If the the current value matches the original, then the value is a regular optimistic update and constitutes normal behavior. If the current value matches neither the new nor the original value, then the field is identified as a conflict (λ) and the framework will attempt to address it using some method $f\left(D'^{\perp}, d^{\top}, d'^{\top}, d^{\perp}\right)$. If f is able to resolve the conflict, then it is cleared and no further special consideration is given to the item. This can be more formally described as

$$\lambda := d'^{\top} \notin \left\{d^{\top}, d^{\perp}\right\} \wedge \neg f\left(D'^{\perp}, d^{\top}, d'^{\top}, d^{\perp}\right)$$

and dictates the nature of the record in the change set. To more clearly associate these rules to the states of an object outlined in table 3.1, table 3.3 is offered.

After reconciliation, if conflicts were present ($\exists\lambda$), Δ' is then compiled and shipped back to the client providing details as to the current state of items submitted and all conflicts found. If there were no conflicts the change set is committed to the DB. In addition, if a PC lock is currently held on p by the requester, the lock is released regardless of the value of b.

Changes are submitted in bulk. Since users are to be able to work while offline, the assumption is that multiple items are edited while offline Thus, it stands to reason that all changes are related and should be processed as an atomic action. Consequently change sets are submitted for updates to the

[2]This also supports backward compatibility as legacy, incongruent, schema for D' may continue to submit and receive state from the application. Something incredibly useful given the pace of software development and growth of mobile access.

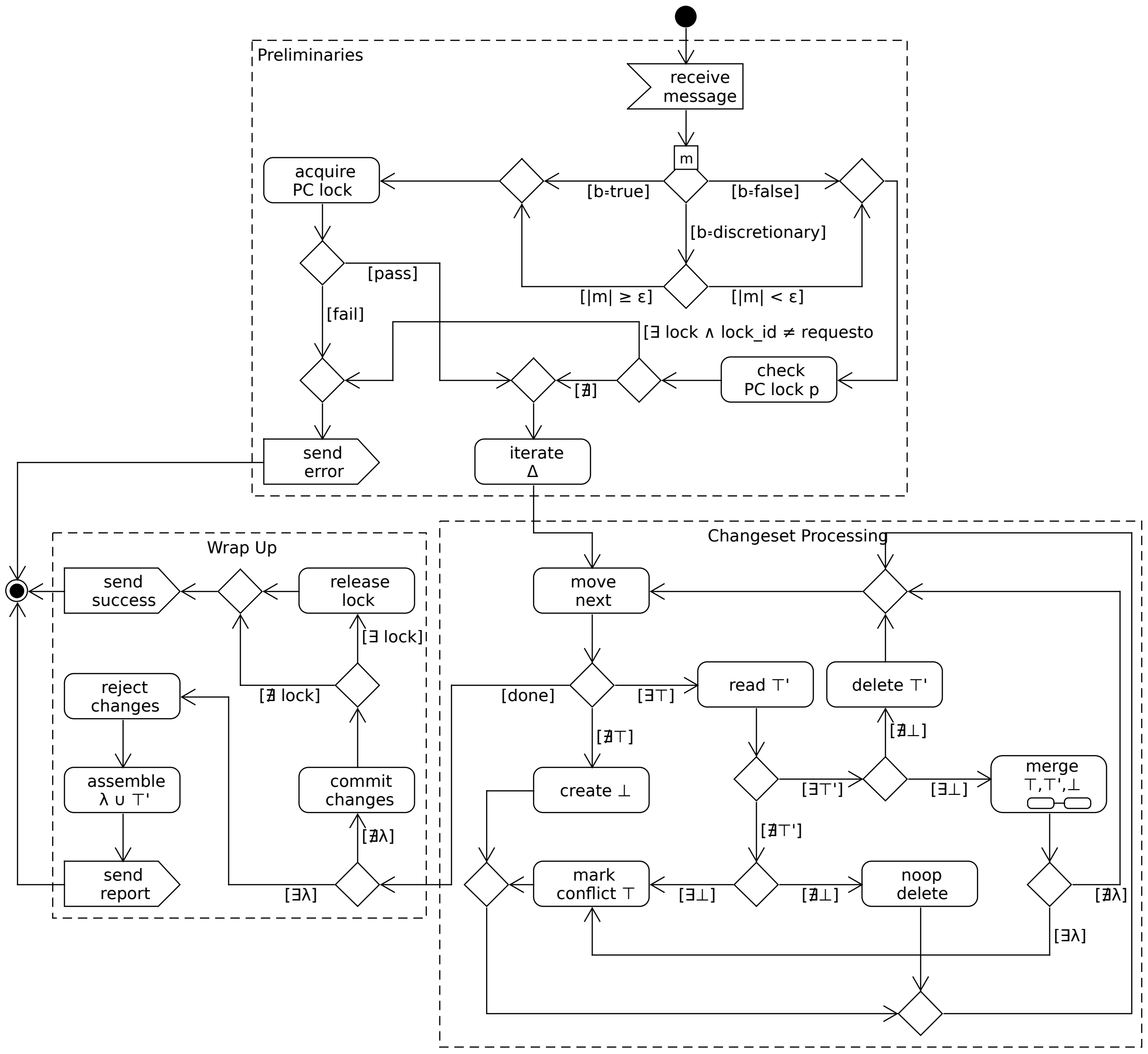


Fig. 3.4: Server processing

Term	Description
p	A partition of data.
m	A message communicating state between client and server.
b	A flag indicating desire to have PC used or that PC is in use.
s	the state of negotiation.
$\top$	Original value.
$\perp$	Modified value.
Δ	A change set communicating changes between client and server.
D	A data item within a change set.
d	A discrete property for a data item.
$'$	Server knowledge.
λ	Conflicting changes.

Table 3.4: Terms Quick Reference

server to be processed as a script of changes to be applied.

Note that the signature of f takes the destination object, the original value, current value, and next value. By the time f is used, $D'^{\perp}$ contains all values as they would be persisted at that moment. This provides one critical piece to the chain which is validation. Rather than blindly accepting values based upon some simplistic hard coded scheme such as last-write-wins or all dirty writes are rejected, business logic may be introduced and the object validated prior to persistence. The system is ignorant to this as any validation error is simply picked up as a conflict in the reconciliation processes. This results from the notion that only valid data was permitted to be persisted in the first place, thus attempting to merge from a valid to an invalid state likely indicates that a conflict occurred. This also permits that new values may be vetted against more than merely last, current, and next. The proverbial door is opened to opportunities for merge strategies which include concepts such as variance, percentages, and tolerance across the state of the object as it will be. In short, intelligent automated merge strategies are possible which may be tailored to a particular purpose. This facet of this approach is what drives the work in general and enables improved conflict resolution strategies.

CHAPTER 4

IMPLEMENTATION

As alluded to in the introduction, the code in this sample is based on the needs of a real world project. The unique needs of this project specifically entail:

- Offline editing of shared data.
- Re-integration of maximum edits with minimal loss of data by any party involved.
- Streamlined work flow for re-integration.

In the project motivating this work, users were expected to survey, measure, and report on assets in the field. These assets may be in remote or poorly connected locations such as sub-basements and high-rise roof tops or regions with no mobile access which often times were under construction. Naturally, they would lack amenities which provide strong network connectivity yet users were still expected to record data within the system from these areas. To further complicate the expected use cases, users are expected to share work. It is unknown in advance what users will touch which records in the database and many may share updates. For example, if two users inspect the same piece of equipment, one may capture details such as serial number, make, and model while the other user may capture details such as voltage, current, and load and capacity. All details are recorded to the same data row but updates between the two should remain in tact. In some respects, providing a more aptly normalized form of the data could circumvent some of the issues encountered here, but then in the real world, such changes to an existing system are often met with strong reserve.

This proof of concept takes the requirements further by providing deterministic yet variable conflict resolution strategies to increase the volume of accepted transactions without imposing micro-management of the data to the end user where possible. This sample implementation is split into two core domains between the client application and server component. The technology leveraged in this solution demonstrates the occasionally connected clients ability to self heal under the self-actuated lock promotion scheme outlined in chapter 3.

4.1 Client Implementation

The client is an HTML5 web application built on AngularJS 1.4.7 and leverages the following extensions to Angular:

- AngularJS Animate
- Angular UI Router
- Angular UI Bootstrap
- angular-moment
- angular-toastr

At its core, AngularJS is a dependency injection framework designed to enable domain driven development using an MV* style pattern on an HTML5 compatible browser. This perspective enables the declaration of non-HTML elements within a web page to be properly interpreted to by the browser and rendered as regular HTML code. Using customized elements, attributes, and CSS classes, one is able to inject new semantics into the development of a web page. This allows for a more focused development effort on the reasoning behind the code and less effort is spent reconciling the application code to the browser.

Each extension to the tool provides specific functionality and is meant to fill some gap in the bridge between application development and the content viewed by a user of the product.

4.1.1 Extensions

AngularJS Animate

The Animate extension provides hooks into the document object model (DOM) for applying visual transitions in the document as it evolves over time. Advancements in web development have moved focus from many disparate pages linked together though HTML into a more consolidated view termed Single Page Application or SPA. Providing feedback to a user is always an important aspect of development as it gives an application a responsive feel and more intuition to a user of the tool. Pushing a web browser to remain resident in a page for a longer period of time removes a type of feedback the user is historically accustomed to in that they lack the experience of leaving a page when moving to the next. The Animate package provides the means to supplement this legacy experience with a more modern, interactive one.

The animate package exposes hooks in the form of HTML attributes and CSS classes to intercept changes in the DOM and apply CSS3 transitions to the changes detected.

Angular UI Router

The normal navigation pattern used in AngularJS, as with many other SPA frameworks, is done through URL mapping. As with any HTML SPA site, the SPA lives where the '#' character in the URL begins. Everything after the hash represents a unique resource within the SPA. Mapping the last half of the path to pages within the SPA allows for the SPA framework to intercept and redirect the content on the page while remaining at the root page of the application. The UI Router project takes this a step further by converting from URL mapping to state mapping.

URL routing is useful as it allows for linking into an application past the landing page, however, inputs into the target zone may be cumbersome to resolve as they must be translated from the URL into stateful objects within the application. The UI Router assumes the responsibility of this translation so that the developer may focus efforts on the destination itself and less on how to get there.

Angular UI Bootstrap

The UI Bootstrap extension is a port of the well known interactive parts of Twitter Bootstrap into the Angular way of doing things. Bootstrap is built using jQuery which is a robust framework which abstracts out the browser specific methods of performing activities into a single unified API. It also provides an extensive suite of utilities which extend the JavaScript language through prototype inheritance. However, jQuery is focused on the DOM and its manipulation, where Angular is focused on the application and its behaviors. Angular uses a more lightweight form of jQuery for achieve the same browser abstraction but without the bloat of focus on the DOM. With this in mind, the UI Bootstrap extension seeks to provide all the functionality of Bootstrap with the same change in emphasis.

Other Extensions

Angular Moment and Angular Toastr are also ports of well established plugins. Moment is a library centered around the rendering of points in time which provides a more friendly, human readable

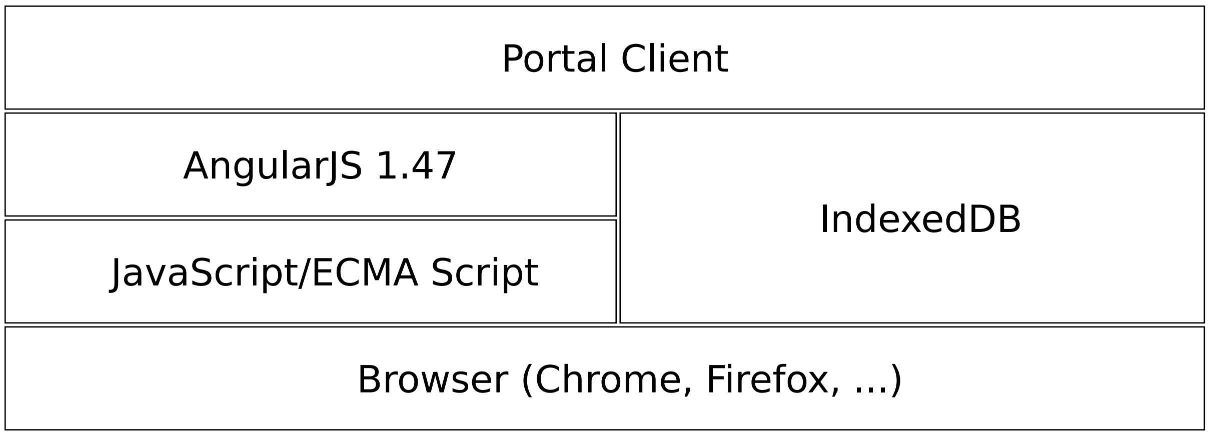


Fig. 4.1: Client Technology Stack

form of a date. For instance, instead of saying “This job was checked out at 2015-11-01 09:12.09365” moment will render “This job was checked out 16 minutes ago.” Angular Moment allows for this behavior in a declarative way such that by simply decorating our time with an attribute, Angular will assume responsibility for the conversion and no further back end coding is required on the part of the developer.

Angular Toastr is a port of the well known toastr javascript framework built on jQuery and provides a friendly, unobtrusive pop up notification system.

4.1.2 Client Application

The documentation for the client documentation is given in appendix A. The full functional code is available in appendix B. Figure 4.1 provides a high level overview of the technologies used on the client. Development of the application was performed in Google Chrome version 47 as well as Firefox 42 (Windows, Linux, and Android variant of each included). The site should function with Safari and Opera as well. Internet Explorer 8+ has an issue with the IndexedDB shim used to enable Safari and Opera which was not addressed. The Android native browser based on Chrome has known issues with offline storage and has also been deprecated by Google in favor of their more complete Chrome browser. As such, no effort was invested in this browser.

The client application is a simple collection of static basic .HTML, .js, and .css files. It is built specifically to empower offline editing of data and the reintegration of that data back at the server in a deterministic fashion. It uses HTML5 application caching provided by the browser to make the application completely available while offline. To ensure data is available as well, it leverages the IndexedDB web storage standard to allow potentially significant blocks of data to be replicated locally.

Cache	Store name	Description
$\top$	asset	Always represents last witnessed state at server.
$\perp$	asset_shadow	Always represents most recent version modified by user.

Table 4.1: Example cache partitioning in client.

In the original project, a single job may have upwards of 5000 assets and each asset may have up to 3 forms to populate with each form containing up to 90 fields. That makes for an upper bound total of approx 1.35M fields required to be available locally on the client.[1] If a user checked out several jobs, the data volume would not fit in the standards sizes available to most modern browsers. Hence, the use of IndexedDB as the storage standard to garner larger storage capacity.

The client operates in two modes, online and offline. The mode is largely transparent to the end user and is mostly independent of the the hosting device's connectivity status. Rather than the client itself being the online/offline determining factor, the status of the job in view dictates the mode of the application. This follows as a consequence of a reactive nature of the application. Specifically, when a user will leave for the field to visit a job site, the user proactively checks out a job. This places the job in an offline state independent of the hosting device and ensures that the user is able to work on the data as it was when he or she left the office. This effectively marks the begin of transaction for the clients work. The client is viewed as reactionary in this model as it does not proactively acquire and cache data locally. This enforces a naturally online state for the client for all data no expected to be manipulated remotely.

On checkout of a job, data flows into the IndexedDB local store and constitutes the $\top$ portion of the local cache. Items are read from this source to be edited while offline. Once an item is edited, the saved version is not written back to the $\top$ but is instead directed to the $\perp$ region of cache which is implemented in IndexedDB as the same table name with a postfix of "_shadow". See table 4.1 for an example. Once an item exists in the $\perp$ region, it is always read from there when presented to the user. The $\top$ region for the item is only accessed when submitting to or receiving changes from the server or when providing context for the user to reconcile data which the server is unable to.

For practical purposes, this implementation is restricted to only the two primary forms of entity utilized, Job and Asset. This allowed for less effort to be invested in the unnecessary comprehensive feature set exposed in the client while still enabling practical demonstration of the components relevant

[1]While 1.35M fields may be made available locally users only interface with and modify a small sub set of these per change set. However, all must be made available as it is unknown in advance what those records are.

to this thesis. Consequently, the client component is significantly stripped down in terms of required API. There are three core components to the client. In Angular speak, they are the job service, offline service, and the asset controller. The services are responsible for shepherding the data two and from the proper sources. The controller hosts the data for the user to edit.

Client Services

The job service speaks with the server for online data and the offline service for offline data. It also knows how to interpret a subset of data from a job for proper routing of its data and the assets attached to it. The primary API methods of this object include:

Abandon	drops all locally cached data for a job. May be used directly by the end user to cancel work and return the job to an online state. Also leveraged by the check in method to clean up after successfully pushing changes to the server.
Check in	compiles a CS as prescribed in chapter 3 and submits it to the server then listens for the server's response. If the server responds with an HTTP status code 204 (no content), the check in was successful and the method calls to abandon to cleanup local data and return to an online state. If the server responds with an HTTP status code of 409 (conflict), this means that conflicts were detected which the server could not reconcile or that the job the user attempted to check in was locked by another user. Should the job be locked by another user, there is no content in the response from the server, otherwise, a CS was returned as prescribed in chapter 3. The client then imports the accompanying CS into the $\top$ portion of the local cache and moves the job to a reconciliation state. Future accesses to the job's assets will now pull data from $\top$ and from $\perp$ for the user to reconcile changes.
Check out	obtains an initialization CS from the server and imports it into the $\top$ segment of the cache then moves the job into an offline state.
Get all	gets all jobs that it can, online and offline. The method by which it renders them is somewhat counter intuitive but makes for more accurate representation of data to the user. Specifically, jobs which are offline are "merged" with their online state if found in both queries. This is the only method which blends data in this way and is useful in that it enables the client to render the current information for the job (such as locked status) while maintaining its isolation from the online state.

Get asset* exposes assets to the client. As assets are children to jobs and job state dictates the online/offline status, the job service exposes the assets as well. This allows the job service to properly route queries based on the state of the job. While the job is online, all traffic is direct routed to the server through standard HTTP AJAX requests. While in offline mode however, this service routes queries to IndexedDB through the offline service.

Offline Service

The offline service's sole responsibility is interfacing with local cache and is implemented atop IndexedDB. This service exposes all data in the local cache using the pattern of promises. All updates and queries issued to this service are tiered. An initial query is issued to the $\perp$ segment. If the item is not found there or if all items are requested, another query is issued to the $\top$ region. If all items are requested, the $\perp$ results are merged with the $\top$ results and where a collision exists on the identity of records, the $\perp$ version is selected. Each query has the capability to bypass this behavior through a `suppressShadow` boolean argument. If the argument is explicitly `true`, then the bubble behavior is suppressed. This enables direct queries and updates to access explicit shadow regions of cache and the return results from a failed check in request to update $\top$.

Web storage is a still new topic and no set standard has been established as the de facto standard. The two main versions available today are IndexedDB and WebSQL. WebSQL has been semi-cut from the running by the W3C though. W3C prefers to have a minimum of two options of implementation for any standard it puts forth. Unfortunately for WebSQL, all major browsers have essentially simply imported SQLite into their browsers to add the missing functionality. Consequently, there is really only one implementation available and thus the W3C has placed a hold on progressing the WebSQL standard until such time as another option becomes available. After several years of being in a holding pattern, the W3C decided to simply move on and elect IndexedDB as the standard with their backing.

Since the W3C moved to back IndexedDB, it has seen some growth in its acceptance, however, it is still not fully supported across all browsers. However, all major browsers support at least one of the two of WebSQL and IndexedDB. As such, a polyfill has been written to expose the IndexedDB API over a WebSQL back end. All of these factors combine to justify this clients implementation of offline caching using IndexedDB.

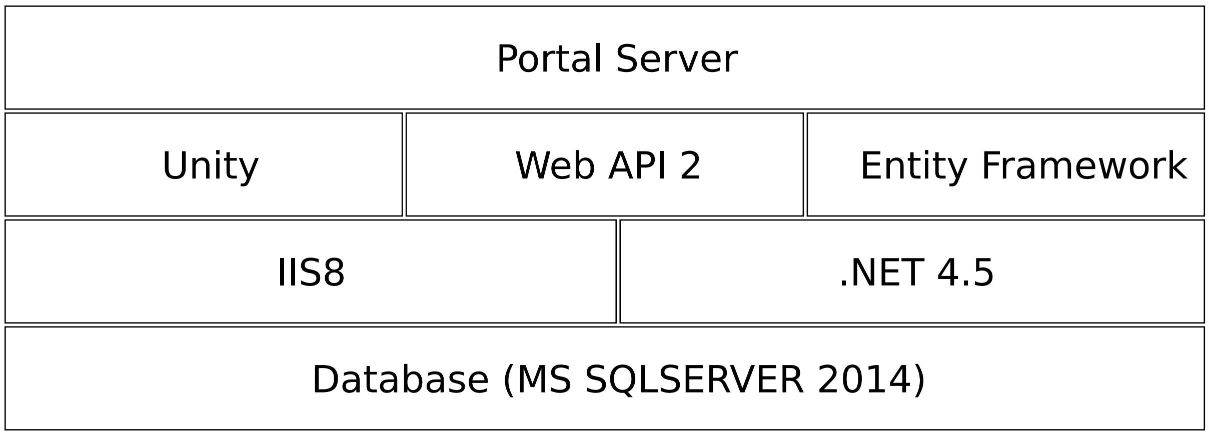


Fig. 4.2: Server Technology Stack

4.2 Server

The server component is built on ASP.NET using the Web API 2 framework. The server documentation is available in appendix **??** while the full source is in appendix C. Figure 4.2 provides an overview of the technologies used in the server's development while figure 4.3 shows the core areas of the system with their intrasolution dependencies and accompanying role fulfilled.

As discussed in chapter 3, the client transaction is independent of the server. This aspect of the client permits the server to run in a stateless manner and allows that the fate of the, potentially many, client transactions be independent of the server's overall state. In turn, this ensures that the client may operate in a transactional mode and the server need not be burdened with anything more than its core function of data access and maintenance. This nature is apparent in the overall design of this implementation. While the server is built on ASP.NET, it actually uses very little code in dealing with anything other than the data.

4.2.1 Web API 2

The Web API 2 standard is a simple REST-ful service layer exposed over the standard HTTP protocol. Thus, any library which supports the very well established HTTP protocol supports the Web API 2 standard. In this framework, a collection of API controllers are provided to expose data to the client.though HTTP requests. The controllers in this sample are limited to minimal functionality to meet the needs of this proof of concept.

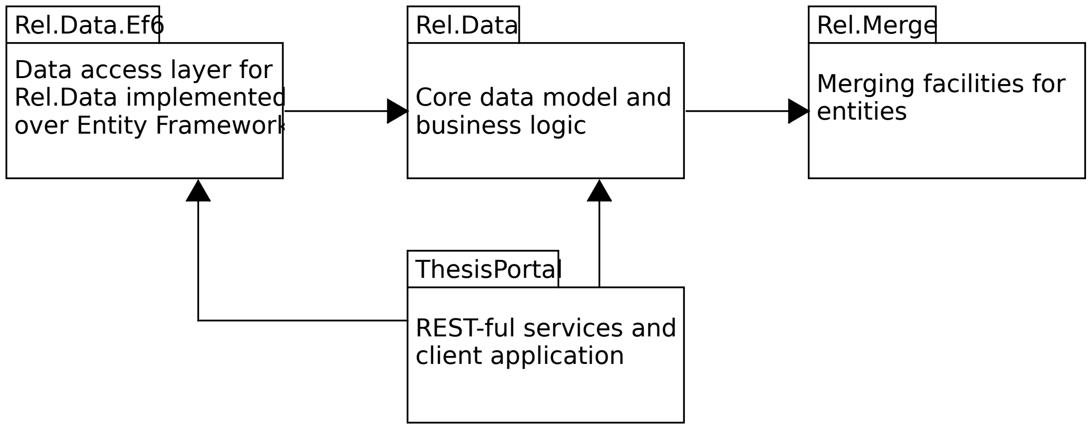

Rel.Data.Ef6
Data access layer for Rel.Data implemented over Entity Framework
Rel.Data
Core data model and business logic
Rel.Merge
Merging facilities for entities
ThesisPortal
REST-ful services and client application

Fig. 4.3: Server Packaging Overview

4.2.2 Authentication Controller

Being a proof of concept, the authentication requirements by this system are very narrow in scope. The only authentication required at this time is that the server know who the user is in order to manage any pessimistic lock the user may be using. This controller is limited to functions to log a user into the system, out of the system, and to identify the user.

Authentication is provided using traditional forms based authentication predicated on an authentication HTTP cookie which the browser manages. In a SPA, it can be a challenge to reconcile the authenticated user with the server, as such, the `Identify()` method is provided to allow the client to query the server for the user credentials mapped to the authentication token cookie in the browser. If the user is logged in but the application lost track of the authenticated user via some behavior such as refreshing the page, the system is able to reestablish the user's session via a query to this method. If the user is not logged in, this method simply returns a standard HTTP 401 unauthorized status code.

4.2.3 Jobs Controller

The jobs controller is the root hub for entry into the product. This controller provides methods to query jobs which the user is assigned to and to get a single job. For the purposes of this implementation, no further access is required as the job is read only to the client.

4.2.4 Assets Controller

This controller is slightly more robust than the jobs controller as it provides methods to get assets organized by a per-defined category such as an assigned service area, all assets on a job, and to update an asset. In this sample implementation, editing is intended to be conducted offline. However, in order to easily produce conflicts in the system to examine reconciliation, the ability to update an asset while online is provided. This simply eliminates the need to checkout a job and make changes while exploring the behavior of the reconciliation process.

4.2.5 Offline Controller

The offline controller is lean and serves only to expose a REST endpoint for the underlying CS processing architecture. It is responsible for governing communication of the CS's and results of CS requests with the client.

4.3 Server Framework

CS's are processed on the server through a framework which operates under the constraints outlined in chapter 3. The framework itself is described here.

4.3.1 Change Set Processor

The `ChangeSetProcessor` accepts requests for batch operations and translates them into data domain operations. When the data layer reports issues with the data, the processor attempts to reconcile them and possibly retry the update. This class exposes only two endpoints which represent the lion share of API into the server.

The read endpoint to the framework is the `BuildInitialChangeSet` method. This method simply reads a partition of data and returns the data items within the partition in CS form. Each item in the resulting CS are flagged as initialization changes which indicate that the recipient should create or set the local cache to hold each item there in. This is in effect simply a batch query method. It is worth noting here that there are no side effect artifacts generated by invoking this method as the server is genuinely stateless with respect to its clients.

The write end point is `Process`. This end point accepts a CS, partition identifier, and a boolean flag to indicate if the system should provide an exclusive PC lock on the partition if the CS fails to be accepted. In a real world implementation, the partition should be implicit and contained within the CS while the partition management would be delegated to a fellow handler. For the sake of expediency this aspect was intentionally not re-factored into this implementation.

Once a CS is received for processing, the first action taken is to replay the actions of the CS into a CS (CS_{DAL}) which is understood by the underlying data access layer (DAL). In this case, Entity Framework 6 (EF) is the underlying provider and the Rel.Data.Ef6 library package in figure 4.3 provides an implementation of the `Rel.Data.IDataContext` interface to facilitate the activity.

Once the CS_{DAL} has been prepared it is validated. If validation errors are found, the system will throw a standard validation exception error message; otherwise, it progresses normally to the next step. Once the CS_{DAL} is prepared and validated, a transaction is started. Late binding the transaction in this way helps to minimize resource contention as the CS is potentially large and may take some time to process. Once in this transaction, a query is invoked to read the header of the partition being updated, the job. The job is then inspected for PC lock status. If the job is locked by another user, the CS is immediately aborted with an error in the form of a `Rel.Data.PessimisticConcurrencyException` being thrown. If the job is exclusively locked by the authenticated user, or the job is not locked by any user, the DAL then executes its CS_{DAL}.

When the underlying DB accepts the CS_{DAL} the system simply commits the transaction. On success but before returning to the caller, the process will inspect the PC status once more. If the authenticated user holds the PC lock, the it will be released before returning a successful status to the caller. If, on the other hand, OC conflicts were detected when executing CS_{DAL}, the DAL throws a `Rel.Data.ConcurrencyException`. The processors listens for this error and handles it distinctly. Upon receiving an error of this class, the processor will first abort the current transaction. Rolling back is important as the CS_{DAL} is only partially written. To not do so would cause the transaction to no longer be atomic or isolated.

After managing the transaction, the CS_{DAL} is dropped. Since the CS was rejected by the DAL, at least one item in the CS has been modified since being read. As such, the entire CS will need to be checked for which item or items are in conflict. Do do so, the DAL will need to expose the proper items, not the ones which were setup in the CS_{DAL}. Consequently, the current state of the CS_{DAL} holds no value any longer. Thus, it is simply dropped before proceeding. If the job is not already locked by the current user but should be, as determined by the CS size ($|m| \geq \epsilon$,) or the user request flag (b), the system attempts to lock the job from within a new transaction. If this step fails, the system aborts the transaction and throws a PessimisticConcurrencyException; otherwise, it proceeds to redress the CS. It is important to note that any manipulation to the PC lock is always done in an independent transaction, never in-line with other changes. This design is intentional as it helps to prevent deadlock scenarios and keeps changes as minuscule as possible, a very important trait since dealing with bulk changes is slow.

CS rejection may occur due to one of three expected causes. The first two have already been established as a conflict in the PC lock and OC conflict of one or more CS items. The other expected cause of CS rejection could be a violation of DAL constraints such as foreign key integrity or attempting to assign a null value where it is not permissible or some other form of business invalidation. These types of errors are assumed to have been previously addressed by the calling code as typical business logic validation. These type of error are mentioned here only to establish their consideration and intentional dismissal from scope as well as why. However, the interfacing with the validation subsystem is plumbed and used in this sample.

At this point it is possible to skip straight to section Reconciliation Change Sets and the approach would be satisfied. However, the fruit of this approach lies not in what it does but in what it enables. Namely, using this approach, the server is able to accept transactions which it would otherwise be required to reject. The Merging section elucidates a mechanism to leverage this approach for more favorable outcomes to transactions more often.

Redressing Rejected Change Sets

By now, the initial change transaction has been closed. If closed due to pessimistic concurrency the system simply allows this exception to bubble out to the caller. In this way, the caller can elect special behaviors for this unique error condition. If the change transaction were closed for concurrency exceptions, the processor moves into the redress phase. In this phase, the submitted CS is inspected and compared to the existing state for each item it contains. Items from the CS are paired with entities read from the DB and handed off to a sub system to attempt to resolve any outstanding conflicts.

In this solution, all access to individual entities is done through a repository pattern. The implementations of the repositories here are provided in the Rel.Data.Ef6 package by the `Rel.Data.Ef6.DbRepository<TEntity, TKey>` class. The repository interface supports two forms of query. It implements `IQueryable<T>` and provides a `GetAll()` method which returns an `IQueryable<T>` object. Any queries exercised against the `GetAll()` method will execute against the DB. All queries run against the repository directly will execute against the local cache of the repository.

To resolve conflicts, the server first starts a new transaction and pulls the current state of all items in the CS from the DB into the local cache of each repository type used in the CS. To do so, it first empties each local cache to discard the CS entities which need to be evaluated. This ensures that there are no errant cache hits while trying to redress each item. Next it executes a query against the `GetAll()` method to pull items of the given type present in the CS. All queries for caching are performed in a single transaction. Pushing these queries into their own transaction serves two purposes. First, it permits interleave in the processing of CS. Secondly, it ensures that the act of redressing a complete CS is done against a single coherent view of the data in the DB, ensuring the isolation property of the overarching business transaction is maintained. After all items are cached, the transaction is closed and the work of redressing commences.

In order to redress divergent changes of items in the CS, the processor iterates over each item in the CS. In this implementation, the concept of a conflicting create operation is rejected. Consequently, there is no need to iterate the repository directly as there can be no item in the repository which is not in the CS. For each item in the CS which is not a create operation, the current item is queried from the repository cache. If the item is not found, the item was deleted and will be handled under the special case logic for hidden delete. Found or not, the result of the cache query is joined with the CS item and the pair is handed off to an `Rel.Data.Bulk.IConflictResolver`.

This solution provides two implementations of the conflict resolver:

- `Rel.Data.Bulk.RejectConcurrentEditsConflictResolver`
- `Rel.Data.Bulk.MergeConcurrentEditsConflictResolver`

The no op resolver simply rejects all conflicts, exhibiting the the same behavior as a system using normal OC with no augmentation. Alternatively, the merge implementation integrates with the merge framework provided by the Rel.Merge package identified in figure 4.3 and is covered in section 4.3.2.

When a CS has completed successfully, if the partition updated has an exclusive lock applied to it, the lock is released. Again, this occurs in its own transaction to minimize potential for dead locks. Also, the system verifies the authenticated user holds the exclusive lock before releasing as it is possible that the CS was accepted under OC lock and before the PC transaction ran, another user acquired a PC lock.

Reconciliation Change Sets

Any CS which is unable to commit and is in a rejected status warrants a reconciliation CS (CS_r) be issued to the client. The CS_r contains the current state of all items submitted in the initial CS. The

Classification	Description
Hidden Delete	The item was deleted from the database after being read from the database.
Dirty Delete	The item has been modified in the database after being read and is marked for deletion in a change set.
Dirty Write	The item has been modified in the database after being read and is marked for update in a change set.
Optimistic Write	The item has not been modified in the database after being read and is marked for update in a change set.

Table 4.2: Classifications of Divergent Data

client is then able to use this new CS_r to generate a new CS for subsequent submission.

Compiling a CS_r is a fairly remedial task. When the system elects to generate such a CS, it begins by rejecting all changes in the data context. This ensures that all items in the new CS reflect the snapshot of data from the DB. The system then iterates over all items in the data context and the original CS. The union of the the items of both sets forms the new CS_r. This CS_r precisely mirrors the CS submitted by the client to the server in terms of the actions requested, except the net result of the actions of CS_r will result in the client's $\top$ cache reflecting the snapshot from the DB.

4.3.2 Merging

Using a framework built on .NET class attribute decoration, rules are applied to allow acceptable amounts of fluctuations in the data which permits relaxation of the definition of incoherent without sacrificing the rigor of its application. This framework supports the three classes of divergent data and four classes of merge operation described in table 4.2. to enhance the check-in process. The Rel.Merge library is small and is focused on the objective of inspecting multiple objects in order to determine a single coherent view of them. This library allows for easily extending any object with new semantics for OC conflict resolution as well as seamlessly adding new rules to the system. It starts at design time, while planning the caching and merging strategies of the solution. Classes are decorated with class and property attributes which derive from `Rel.Merge.Strategies.MergeableAttribute`. An example implementation of class decoration and attribute usage is given in listing 4.1. The `MergeableAttribute` serves as the base sentinel for opting into this OC conflict resolution model. It exposes a single method `Merge<T>` which takes a single `MergeAction<T>` as its argument See figure 4.4 for a summary of its public signature. The merge action doubles as the result of the operation which helps in performance

```
    //[DirtyDelete]
    //[HiddenDelete]
    [LastWriteWins(false)]
    public class Asset
    {
        public int Id { get; set; }

        public int JobId { get; set; }

        //[DecaySpanMergeable("0.00:00:00.3", "0.00:00:00.3")]
        //[StepMergeable(true, 0.3)]
        //[LastWriteWins]
        public double? MaximumAndMinimumDecay { get; set; }

        //[DecaySpanMergeable("0.00:00:00.3", "0.00:00:00.3")]
        //[StepMergeable(true, 0.2)]
        public double? MaxMinDecayWithStepAndTol { get; set; }

        public double MinimumDecay { get; set; }
}
```

Listing 4.1: Example class decoration for merging.

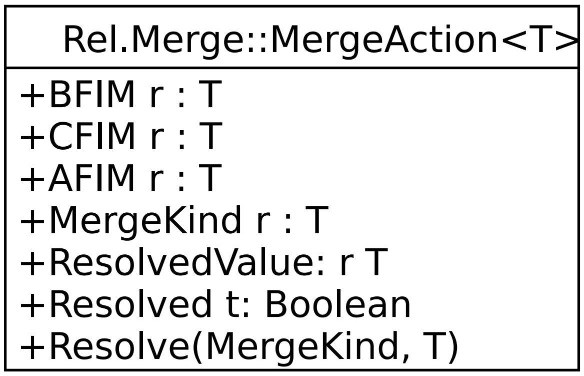


Fig. 4.4: MergeAction<T> public interface

as well as clarity of code. By reusing the same object, it is possible to provide tiered merge strategies as the one object contains sufficient information to see the following.

- Where the item came from (BFIM)
- Where the item is going (AFIM)
- What the item is (CFIM or Current)
- The kind of conflict which is being resolved (see table 4.2)
- If the item has been resolved

Has Value				
BFIM	CFIM	AFIM	Implication	Is Conflict?
Yes	Yes	Yes	Standard OC Update	Maybe
Yes	Yes	No	Standard OC Delete	Maybe
Yes	No	Yes	Hidden Delete	Guaranteed
Yes	No	No	Hidden and Dirty Delete	Guaranteed
No	Yes	Yes	Not possible. No BFIM indicates create operation	N/A
No	Yes	No	Not possible, entity was not in CS	N/A
No	No	Yes	Standard Create Operation	No
No	No	No	Not possible. No BFIM indicates create operation	N/A

Table 4.3: Implications of Values Present and Absent in MergeAction

- What the proper course of action to resolve the conflict is, if any need be taken by the calling context

The presence of BFIM, CFIM, and AFIM combine in interesting ways and table 4.3 provides a summary of interpretation for them. The merge framework refines the granularity of a commit from a row, block, tuple, set, or entity level to a field or value level. As such, every value of every item in the CS is inspected for consistency and ability to merge[2]. In situations where a CS is large, this could be a very costly process. To abate performance concerns, this library provides some unique facilities in this area which are justified in the following chapter over evaluation.

Invoking Merge

The Rel.Merge framework's entry point is a single type `Rel.Merge.MergeOperation` with a single method `IMergeResolution<T> Merge<T>(MergeKind, T, T, T)`. See figure 4.5 for public interface definition. This method takes inputs reflecting the `MergeAction<T>` above and returns a resolution object containing the resolved value and an indicator as to whether the resolution was successful as well as the action needed, if any, to be taken by the calling code. This type and method serves one purpose which is to scaffold a pseudo proxy type for merging and execute it. However, the proxy which gets generated is different in nature than those typically used.[3] The proxies generated here are wholly owned and managed by the Common Language Runtime (CLR) environment, guaranteed to be single-

[2]All items and values are tried until any one is found to not be mergeable. If any are not mergeable, the system immediately stops merging.

[3]This solution does use traditional dynamic proxies as the Entity Framework 6 and serialization assemblies use them for their needs.

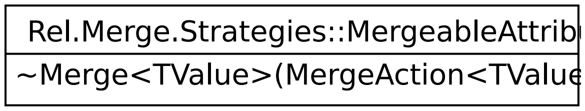

Rel.Merge.Strategies::MergeableAttrib
~Merge<TValue>(MergeAction<TValue

Fig. 4.5: MergeableAttribute public interface

ton definitions, and mapped purely on the type which is proxied as the generic `T` argument without the need for lookup. The only limitation of these proxies is that the properties to be proxied on the given type support both read and write operations. The actual proxy is the `Rel.Merge.MergeOperation<T>` class, not to be confused with the `Rel.Merge.MergeOperation` class mentioned previously. The motivation of the naming convention is intentionally obscured as the non-generic version is the public surface area and should be viewed as the underlying generic version which is internal to the library only. This proxy via generic form is distinct from the traditional proxy implementations because it does not inherit and override. Rather, it wraps access to the properties dynamically through compiled LINQ expressions. This implementation's performance is on par with the conventional native proxy and is reviewed for performance in chapter 5.

The merge starts with the call to the `Merge` method of the `MergeOperation` class. This method creates an instance of `MergeOperation<T>` and hands its the arguments down. The complex work occurs here, when the type is instantiated. This type has a static constructor which performs all the initialization for the dynamic proxy generation. Leveraging statics in the generic definition and the static constructor creates a code path by which CLR does all the heavy lifting of maintenance of the proxies. This also establishes the one time load and initialization of the types in a lazy loaded fashion. The only down side is that this type is not self healing as it can never be unloaded without restarting the process. In practice, this shows to be a negligible penalty. When this type is created, the constructor inspects the generic argument `T` for class level attributes and properties. The property inspection constitutes the dirty work of this process.

Property Merge

Step one of the property inspection seeks out fields used in optimistic concurrency check. This solution only supports a time stamp field but it will not take much work to extend this into other forms of concurrency checking. If no OC control fields are found then the type initializes to a chaos mode in which everything is always last write wins. This behavior mirrors that of any system which does not use concurrency control. Assuming OC fields are present they must be readable or an exception is thrown as this is considered developer error. These fields are then wrapped with a special lambda expression which compiles to a function which evaluates the equality of these fields between two instances of the given type. This method is then saved off to a static field in the class for later consumption.

Next, all non-OC but control fields are identified. These would be fields which should not be merged and should be treated as read only. This includes identity fields and complex properties used for activities like navigation. These fields are partitioned from the remaining fields and stowed away, they do not participate in merge.[4]

The third kind of property inspection partitions all remaining properties into those decorated for merge and those which are not. Those which lack any sort of merge attribute get implicitly decorated with a `Rel.Merge.Strategies.LastWriteWinsMergeableAttribute` which is configured as follows. If any property of the given type is decorated with a merge attribute, the implicit attribute is configured such that last write always wins. If, however, no merge attributes exist in the type, the implicit attribute is configured to reject all writes. The motivation for this behavior is that the presence of merge attributes implies that the developer has considered merge for the given entity type. Since it has been considered, the absence of an explicit attribute for a given property must be deliberate. In such a case, if the implicit attribute took on a behavior of rejecting writes, the attributes which had been explicitly applied would never work as at least one property would always fail to merge. Thus, while somewhat counterintuitive in description here, the behavior when consuming the framework is actually the behavior of least surprise. Finally, all properties which are not control properties now have associated merge attributes. LINQ expressions are now created and compiled to invoke the attributes for later merging.

The ability to merge is determined by the result of a call to the `Merge(MergeAction<TValue>)` method of the `MergeableAttribute` class. If successful, the request's result will be set to a `MergeActionResult` (see table 4.4 for a listing of values) with the `Resolved` bit set (an odd integer value). This is a bit oriented field which may contain multiple values, though only the resolve with one other should pass a sanity check. If any action is need to be taken by the caller, an additional bit field will be set to signal the necessary behavior. The classifications of merge are supported and handled independently. Items of a hidden delete or dirty delete nature lack a critical component for merge, specifically the CFIM or the AFIM respectively. Therefore, the properties of these kinds of merges have no basis of comparison. As such, the property level merge described below is bypassed for these and only the top level type based merge is performed. For items of the OC update or dirty update nature, the following process is used.

[4]These fields are assumed to be managed by the calling context as it knows their true purpose.

Result	Mask	Indication
Unresolved	0x00	Mere was unsuccessful.
Resolved	0x01	Merge was successful and no further action is needed by the caller.
Delete	0x02	Merge result must be deleted from the database to complete the merge.
Create	0x04	Merge result must be created in the database.
Update	0x08	Merge result must be marked as an update.

Table 4.4: Merge Result Bit Field Flags

Before beginning the costly task of property merge, the first step calls the OC comparator function defined above, passing in the BFIM and the CFIM of the merge action. If this function returns true, the item constitutes standard OC behavior with a current entity. As such, the operation is assumed proper, not needing to be merged. Thus, a merge result is immediately returned with only the resolved bit set. If this function should return false however, the merge is invoked at the object level and pushed to the property level. If all mergeable properties merge successfully, the type is implicitly merged successfully. To avoid issues of inconsistent merges at the property level, a special property wrapper merge is used. This wrapper uses a callback to actually execute the merge. Only if all properties can merge, is the callback executed to apply the changes. If not all properties merge, the callback is ignored and the type is inspected for a `MergeableAttribute`. If one is available on the class definition as a whole, it is tried and its result is used instead.

After merging, if all items in the CS successfully merged they are all validated again. Since data may have changed on these items, the merge could have produced invalid objects. Before attempting to talk with the DB again, all items are validated to reduce the DB transaction duration. If all new states are valid, a new transaction is started. In this third[5] transaction, the merged $\text{CS}_{\text{DAL}}$is attempted. If successful the transaction is committed and the resulting code flow is the same as if the initial attempt had been accepted and the approach is complete. If this attempt fails, either due to validation or concurrency, a reconciliation CS is compiled to be sent back to the client.

[5]This may be the fourth transaction, depending on whether a PC lock was acquired in this processing session.

CHAPTER 5

EVALUATION

Testing of this approach and performance evaluation is not easy to conduct as the process is designed to be user centric. However, several tests have been developed to provide perspective on the overall performance and are reviewed here.

5.1 Property Proxies

When considering the performance of this approach, if N represents the number of items in a change set, and P_i represents the number of fields for a given item i, then the approach ranges between $\Omega\left(N\right)$ and $O\left(6N + \sum_{i=1}^{N} P_i\right)$ for an accepted CS and $O\left(8N + \sum_{i=1}^{N} P_i\right)$ for a rejected one. The lower bound originates from the best case scenario where a CS is accepted in whole on the first attempt to save it as fully OC clean. OC clean meaning all items have not been modified since being read. The worst case scenario represents a CS requiring merging where all entities contained therein have been modified and none have been deleted. For the worst case scenario, $6N$ contains:

- The initial attempted save
- Flush of cache
- Enumeration of CS to build cache query
- Processing of cache query
- Translation of cache items to a dictionary for $O(1)$ lookup
- The enumeration of all items in the CS to join with cache lookup
- The final type level merge after the last property merge fails

The summation represents the field level merges required per item. There are other factors not accounted for in these formula including another $O(N)$ for the the DAL enumeration of CS$_{\text{DAL}}$ to effect the changes and various other constants regarding control flow and the $O(1)$ operations for PC lock maintenance. While relevant, these metrics have little bearing on the over all performance or are negated by the fact that they have fixed costs. The rejected worst case adds another $O(2N)$ for the reconciliation change set generation.

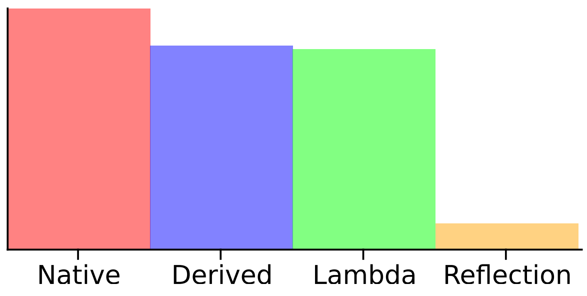
As O highlights, the dominate operation of this implementation while merging is the field level merge operation. Merge is implemented so as to be generic in its application and attempt to merge any object type it is given. This allows for the most reuse of the framework. In a CS which is large, the merge may be called many times. Taking into consideration the worst case scenario from above where there are 1.3M fields to potentially merge[1], the performance of this operation is critical.

In the .NET framework, reflection is, and has traditionally been, used to acquire details from an object when the properties and methods of it cannot be known at design or compile time. Unfortunately, the reflection framework has a reputation for being slow. In the context of refining merge operations, this poses significant concern to the overall performance of this system. A common place practice to avoid redundant reflection is to produce dynamic assemblies which expose dynamic proxy data types that wrap the target objects, extend it with some specific functionality, and link it back into the framework which generated it. These dynamic assemblies are compiled and thus performance for them is on par with the native code written by a developer. However, this approach has several limitations, the more salient of which include:

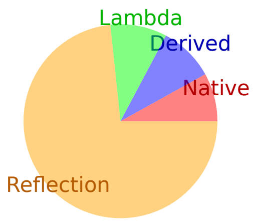
- Classes must be inheritable
- Properties and methods must be overridable to be proxied
- Fields cannot be proxied as they can not be overridden[2]

In practice, it is best to write code which is explicit and unambiguous. Decorating a class member as virtual implies that the object is meant to be inherited and overridden. When using dynamic proxies, the intent of overriding can become obfuscated as it is unclear as to whether the property is overridable for the purpose of proxy or because it is meant to be overridden by the developer. Furthermore, the business of generating dynamic proxies can be somewhat cumbersome and a map between type and proxy type must be maintained. For these reasons, the Rel.Merge library avoids dynamic proxies and assemblies in the conventional sense and a different approach to this problem is used.

[1]While academically this is true, the scenario which spawned this work only touches a fraction of those in a single change set.
[2]The implementation here doe not proxy fields either, however, extending it do so is a trivial task.



(a) Number of Operations



(b) Time per Operation

Fig. 5.1: Runtime performance of property read methods.

This implementation uses a unique approach to solve the problem of performance on read and write operations of an object. A pertinent difference between the requirements of the merge framework and other frameworks which use dynamic proxies is that the objects this framework uses are given to it. The framework does not return instances of them. This enables the system to not worry about inheritance of its proxies for external consumption. As such, the proxy is closed to the outside behind the `Rel.Merge.MergeOperation` type. Leveraging these key characteristics of the system, the implementation uses compiled dynamically generated lambda expressions. Since the expressions become compiled, they become native code which puts them on par with the conventional proxy approach. The generation and concept here is similar to that of dynamic assembly and proxy without the micromanagement of the .NET AppDomain or explicit assembly. Using the compiled expressions, .NET assumes responsibility for the management of the dynamic type information.

To evaluate the effectiveness of this solution, several tests were conducted on performance of this distinct trait. Figure 5.1 presents the findings in graphical form. The data given represents the volume of read operations which are possible within a one second interval under each method presented above as well as relative comparison of the time taken to complete one read operation. Table D.1 of appendix D provides the raw numerical values resulting from running the tests repeatedly over a 30 minute window. Each iteration is run for one second at a time, and a count of read operations which occurred in that window was captured.

Native property read is what the developer would typically write if the type is design time known. An example of its usage is

```
value = obj.Property
```

Override method involves derived types and is what the net result of a dynamic assembly outputs.

However, the discrepancy in performance results from the override which calls to the base implementation. An example of its usage is

```
value = derivedObj.Property
```

Lambda expressions are compiled to native code and the intermediate language (IL) code generated is nearly identical to that of override. The lambda can be expressed in C# as

```
lambda = obj => obj.Property
```

An example of its usage is

```
value = lambda(obj)
```

Reflection is implemented by acquiring the get method of a property as the following C# code does

```
reflected = obj
    .GetType()
    .GetProperty(Property)
    .GetGetMethod()
```

An example of its usage is

```
value = (type)reflected.invoke(obj, null)
```

5.2 Change Set Acceptance

A comparison of the accepted (committed) to the returned or rejected (rolled back) CS under the approach from chapter 3 is presented here. Testing followed the basic pattern of multiple users concurrently producing a CS from known data and attempting to commit it. Every test is run in parallel between the merge framework outlined in chapter 4 and standard OC control which rejects concurrent updates outright. Each strategy for conflict resolution operates on an independent partition of data. While the item types are the same between them, there are no overlapping records to pose conflicts. The only interference they can give each other is by way of DB level table locks applied during create, update, and delete operations. In the context of this exercise this relationship works to the advantage of these tests since, from an algorithmic stance, they do not influence each other but the load imposed between them forces resource contention in processing. This has the added benefit of greater simulated load without impacting the code control flow. Each test iteration is repeated with five minute runtime intervals under the constraints outlined as follows.

.NET performance counters were injected into the code to capture to following details per conflict resolution strategy:

- Total number of CS accepted into the DB
- Total number of CS redressed
- Total number of CS returned
- Total number of CS aborted
- Total number of items accepted
- Total number of items redressed
- Total number of items returned
- Total number of items aborted
- Average number of items per CS with a sampling period of 5 seconds

The variations in test fell along the following inputs:

- Initial test pool item count
- Minimal CS item count
- Maximal CS item count
- Percent chance the user would request an explicit PC lock under the approach outlined in chapter 3
- Percent chance that a conflict is guaranteed for the initial save of CS
- Percent chance an item would be deleted
- Percent chance a CS would introduce new items to the pool

For the purposes of testing, the items test pool is intentionally repressed in volume to increase the natural chance of conflicts under the normal on line control flow as it is the conflicting scenarios which are of interest here. In the absence of conflicts, the approach above is moot.

The user count is also deliberately inhibited to a maximum of five concurrent users selected at random to work a resource pool. Considering the intended use case, the assumption is made that there will not be a large contingent of users attempting to concurrently submit bulk change sets on the same partition of data. Furthermore, the underlying EF DAL used in this sample, while capable, is not terribly efficient at processing large CS. As such, the user load can not creep too high with the CS in tow or the workload becomes unreasonable for both the test server and the DB. In a production setting, this issue can be remedied with detailed evaluation of the query and command batches necessary for the application. Such requirements are not necessary at this time for validating the concept so the artificial constraint is used instead.

Though these tests are emulating multiple users concurrently accessing the data, there are no guarantees as to when their CS may overlap in the correct sequence so as to produce a conflict. To increase the chances of a conflict, an optional setting is used to determine a lower bound probability that a conflict will be generated for any given CS. The conflict is forced by modifying the time stamp value of the BFIM of an item in the CS, thus making the item appear as though it represents an earlier version. Since it only takes one item to be in conflict to cause the entire CS to be in conflict, this change suffices. Also, this is an efficient way to impose a conflict which most naturally simulates the target domain.

To generate a CS for testing, the desired size between the input minimum and maximum is calculated first. A query is then executed to pull that many items into view. This query orders the items by an arbitrary field which is modified with a random value on success. This helps to randomize the items selected for non-deterministic outcomes and collisions. Once the items are available, they are enumerated and converted into change items which are marked as dirty for update or delete determined by the configured input. Next, the decision to create new items is made, satisfied, and the union of the result with existing CS items is returned. Finally, the last decision made before invoking the actual test performs the random conflict enforcement.

Similarly to the lower bound for conflicts, another parameter is used to determine an upper bound probability that the user will make a formal request for PC locking in the event of failure. This user opt-in for PC locking may be disabled by initializing the Required PC % input parameter to zero. However, doing so will not preclude the server from electing to lock in the event of CS rejection and message size verification.

5.2.1 Metrics Captured

Total	reports the total number of change sets encountered.
Accepted	reports the number of change sets which have been accepted and integrated into the DB.

Returned	reports the number of change sets which have failed to be redressed and generated a return reconciliation change set.
Aborted	reports the number of change sets which were abandoned without attempting to execute them due to a conflicting pessimistic lock.
Redressed	reports the number of change sets which were redressed then accepted.[3]
Total#	reports the number of items in all change sets encountered.
Average#	reports the average number of items per change set through the course of the entire run.
Accepted#	reports the number of items in all change sets which were accepted.
Returned#	reports the number of items in all change sets which were returned.
Aborted#	reports the number of items in all change sets which were abandoned.
Redressed#	reports the number of items in all change sets which were redressed.

5.2.2 Metrics Computed

Acceptance	is the dominate target metric in this work and is the basic ratio of accepted change sets over the total number of change sets processed.
Rejection	is the secondary target metric in this work and is the basic ratio of items returned over the total item count.

5.2.3 Test Change Set Acceptance in the Face of Conflicts

To test the behavior of the approach in the presence of conflicts, a series of tests were conducted with increasing commonality of conflicts in each iteration. The following series of tables and graphics illustrate the ability of this approach to achieve near last write wins CS acceptance while retaining concurrency control over the data. The tests conducted were organized such that the changes imposed on data items resulted in valid items. This eliminates user error as a factor in the inspection of the quality of the approach. In addition, existential changes were permitted to data to vet the overall behavior of the system but dirty and hidden deletes did not force conflicts. This was done to allow organized steps in conflict escalation and ensure that the natural occurrence of conflicts did not over

[3]Redressed is a subset of accepted.

inflate the conflict ratio in consecutive test iterations. Furthermore, the type is designed this way to closely emulate a last write wins policy. The final goal that these tests demonstrate is that the tolerance in concurrency is fully customizable and can be on par with last write wins without opt-ing for it entirely.

Table (a) of each scenario detail offers the input configuration using the parameters outlined above. Each (b) table offers a summary of the pertinent attributes of each run necessary to understand the potency of each method and how it fairs relative to the other under the given environment. Metrics on both CS and the items within the CS are provided as they are distinct notions and one does not necessarily imply detail about the other. For example, in scenario 1, the acceptance rate and total CS processed of the baseline OC approach is higher than that of the approach above, however, the approach above processed 318 more items. Independently, they would be merely interesting details. However, the combination of these data points informs that the approach should be expected to be in conflict more often, and indeed it is.

The process begins with the a bottom case light load and is intended to gain perspective of the overall approach as iterations progress. In this first round, the guarantee of conflict and likelihood of PC inputs are both zero percent. Consequently, the performance of each is commensurate with the other. This is a direct consequence of the nature of the approach in that the default mode of operation functions as one of standard OC behavior.

With respect to auditing the approach's impact to the downstream bandwidth, both flavors presented here utilize a higher volume than a traditional model would. In a traditional model, the downstream bandwidth would be limited to the rejection status and possibly a message explaining the reason for rejection. In the results presented here, both the baseline and approach double the update request as a query operation in the event of CS rejection. The response is naturally exaggerated in size as a result. Determining downstream bandwidth impact is done through the Rejection metric. The value presented for this metric actually constitutes an expected one to one request to response situation. Thus, in scenario 1 where the baseline rejected 3.17% and the approach rejected 5.23% of the items each received, it can be said that the approach behaved 65.12% less efficiently than the baseline and that 5.23% of all items it receives will result in downstream traffic being generated with magnitude approximately equal to the incoming request. In contrast, the results of scenario 10 demonstrate the overall downstream bandwidth impact of the approach to be 98.65% better than the baseline.

In reviewing the fate of the items within each change set, Clean refers to the item being accepted on the initial commit attempt with no conflicts at all. Any item which is accepted into the DB coming from the RejectConcurrentEditsConflictResolver may only be accepted when clean.

Scenario 0

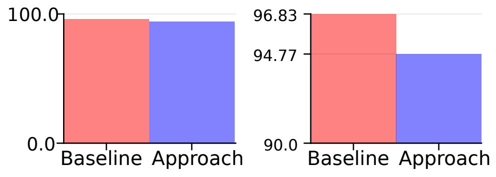
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	0%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

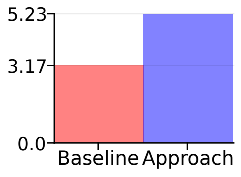
Metric	Baseline	Approach	$\Delta\%$
Acceptance	96.83%	94.77%	-2.13%
Rejection	3.17%	5.23%	65.12%
Total	1,863	1,855	-0.43%
Accepted	1,804	1,758	-2.55%
Conflicts	59	137	132.20%
Actual Conflict %	3.17%	7.39%	133.20%
Returned	59	96	62.71%
Aborted	0	1	-
Redressed	0	40	-
# Average	37.984	38.457	1.24%
# Total	70,641	70,959	0.45%
# Accepted	68,344	67,107	-1.81%
# Returned	2,297	3,805	65.65%
# Aborted	0	1	-
# Redressed	0	1,561	-

(b) Result Metrics

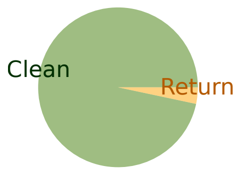
Table 5.1: Scenario 0 Details



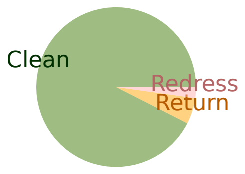
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.2: Scenario 0

Scenario 1

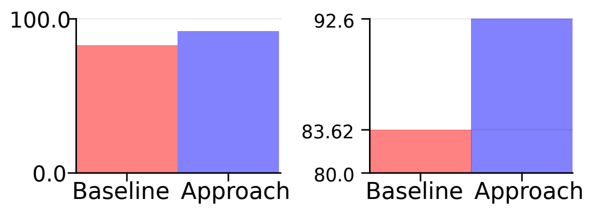
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	10%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

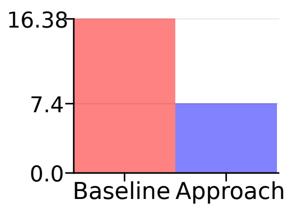
Metric	Baseline	Approach	$\Delta\%$
Acceptance	83.62%	92.60%	10.74%
Rejection	16.38%	7.40%	-54.81%
Total	1,416	1,418	0.14%
Accepted	1,184	1,313	10.90%
Conflicts	232	279	20.26%
Actual Conflict %	16.38%	19.68%	20.09%
Returned	232	105	-54.74%
Aborted	0	0	0.00%
Redressed	0	174	-
# Average	38.029	38.144	0.30%
# Total	53,909	54,208	0.55%
# Accepted	44,822	49,903	11.34%
# Returned	9,087	4,305	-52.62%
# Aborted	0	0	0.00%
# Redressed	0	6,681	-

(b) Result Metrics

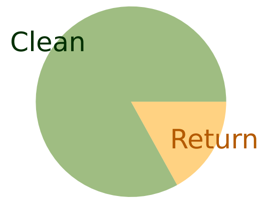
Table 5.2: Scenario 1 Details



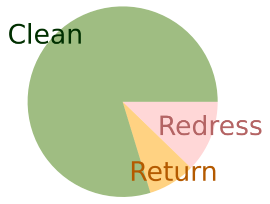
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.3: Scenario 1

Scenario 2

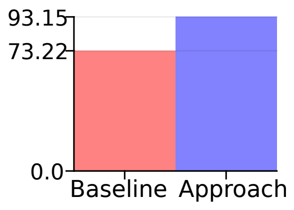
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	20%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

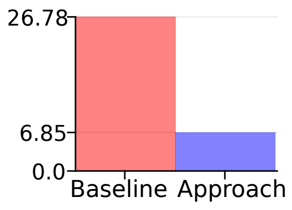
Metric	Baseline	Approach	$\Delta\%$
Acceptance	73.22%	93.15%	27.22%
Rejection	26.78%	6.85%	-74.42%
Total	1,318	1,328	0.76%
Accepted	965	1,237	28.19%
Conflicts	353	394	11.61%
Actual Conflict %	26.78%	29.67%	10.77%
Returned	353	91	-74.22%
Aborted	0	0	0.00%
Redressed	0	303	-
# Average	37.785	37.410	-0.99%
# Total	49,263	49,307	0.09%
# Accepted	35,849	45,818	27.81%
# Returned	13,414	3,489	-73.99%
# Aborted	0	0	0.00%
# Redressed	0	11,400	-

(b) Result Metrics

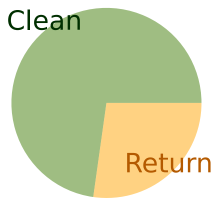
Table 5.3: Scenario 2 Details



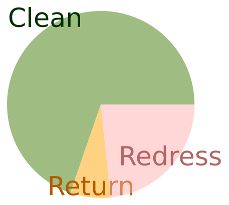
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.4: Scenario 2

Scenario 3

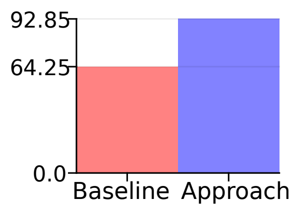
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	30%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

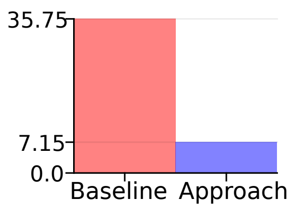
Metric	Baseline	Approach	$\Delta\%$
Acceptance	64.25%	92.85%	44.50%
Rejection	35.75%	7.15%	-79.99%
Total	1,228	12,30	0.16%
Accepted	789	1,142	44.74%
Conflicts	439	443	0.91%
Actual Conflict %	35.75%	36.02%	0.75%
Returned	439	88	-79.95%
Aborted	0	0	0.00%
Redressed	0	355	-
# Average	38.396	37.999	-1.04%
# Total	46,683	46,438	-0.52%
# Accepted	30,100	42,909	42.55%
# Returned	16,583	3,529	-78.72%
# Aborted	0	0	0.00%
# Redressed	0	13,413	-

(b) Result Metrics

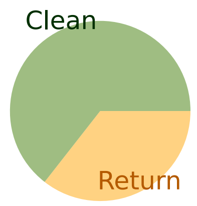
Table 5.4: Scenario 3 Details



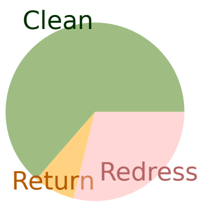
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.5: Scenario 3

Scenario 4

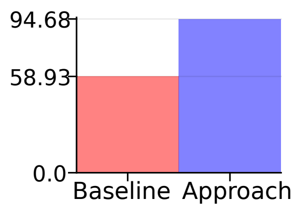
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	40%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

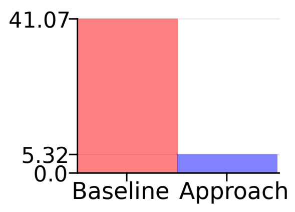
Metric	Baseline	Approach	$\Delta\%$
Acceptance	58.93%	94.68%	60.65%
Rejection	41.07%	5.32%	-87.04%
Total	974	1,127	15.71%
Accepted	574	1,067	85.89%
Conflicts	400	541	35.25%
Actual Conflict %	41.07%	48.00%	16.89%
Returned	400	60	-85.00%
Aborted	0	0	0.00%
Redressed	0	481	-
# Average	37.903	38.471	1.50%
# Total	36,813	43,404	17.90%
# Accepted	21,759	40,973	88.30%
# Returned	15,054	2,431	-83.85%
# Aborted	0	0	0.00%
# Redressed	0	19,380	-

(b) Metrics

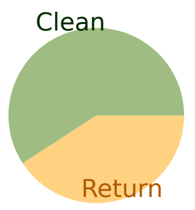
Table 5.5: Scenario 4 Details



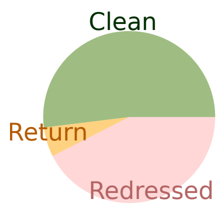
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.6: Scenario 4

Scenario 5

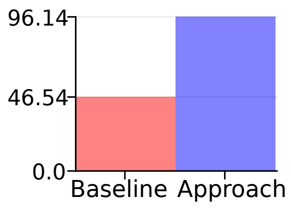
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	50%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

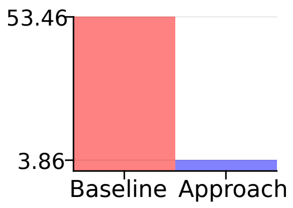
Metric	Baseline	Approach	$\Delta\%$
Acceptance	46.54%	96.14%	106.56%
Rejection	53.46%	3.86%	-92.78%
Total	593	1,192	101.01%
Accepted	276	1,146	315.22%
Conflicts	317	662	108.83%
Actual Conflict %	53.46%	55.54%	3.89%
Returned	317	46	-85.49%
Aborted	0	0	0.00%
Redressed	0	616	-
# Average	37.580	38.081	1.33%
# Total	22,076	45,211	104.80%
# Accepted	10,340	43,309	318.85%
# Returned	11,736	1,902	-83.79%
# Aborted	0	0	0.00%
# Redressed	0	23,146	-

(b) Metrics

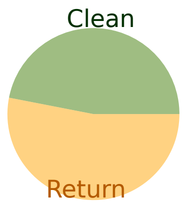
Table 5.6: Scenario 5 Details



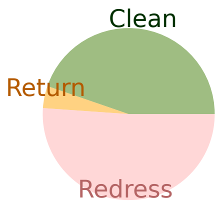
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.7: Scenario 5

Scenario 6

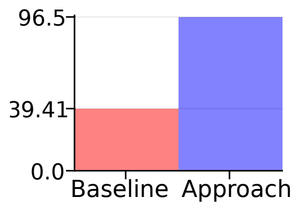
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	60%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

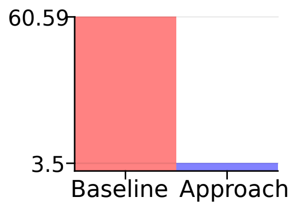
Metric	Baseline	Approach	$\Delta\%$
Acceptance	39.41%	96.50%	144.85%
Rejection	60.59%	3.50%	-94.23%
Total	1,228	1,230	0.16%
Accepted	484	1,187	145.25%
Conflicts	744	789	6.05%
Actual Conflict %	60.59%	64.15%	5.88%
Returned	744	43	-94.22%
Aborted	0	0	0.00%
Redressed	0	746	-
# Average	37.952	37.653	-0.79%
# Total	46,400	46,263	-0.30%
# Accepted	18,127	44,565	145.85%
# Returned	28,273	1,698	-93.99%
# Aborted	0	0	0.00%
# Redressed	0	28,325	-

(b) Metrics

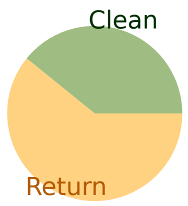
Table 5.7: Scenario 6 Details



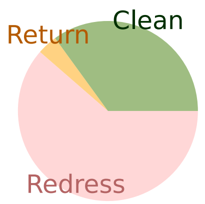
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.8: Scenario 6

Scenario 7

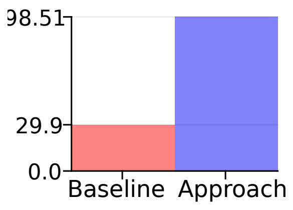
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	70%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

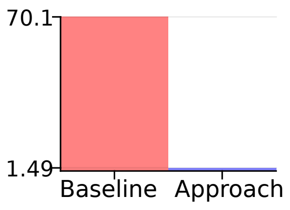
Metric	Baseline	Approach	Δ%
Acceptance	29.90%	98.51%	229.51%
Rejection	70.10%	1.49%	-97.88%
Total	873	874	0.11%
Accepted	261	861	229.89%
Conflicts	612	613	0.16%
Actual Conflict %	70.10%	70.14%	0.05%
Returned	611	13	-97.87%
Aborted	1	0	-100.00%
Redressed	0	600	-
# Average	38.137	37.841	-0.78%
# Total	33,179	33,437	0.78%
# Accepted	9,911	32,957	232.53%
# Returned	23,235	480	-97.93%
# Aborted	0	0	0.00%
# Redressed	0	23,027	-

(b) Metrics

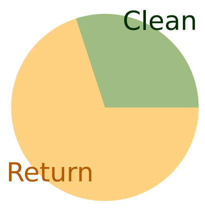
Table 5.8: Scenario 7 Details



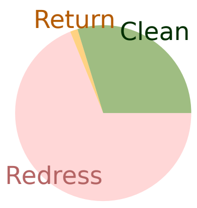
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.9: Scenario 7

Scenario 8

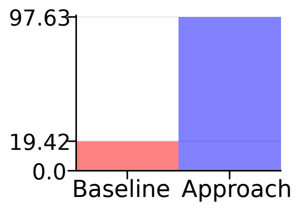
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	80%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

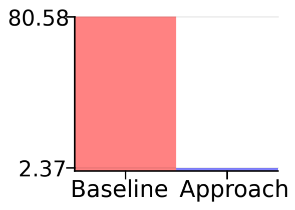
Metric	Baseline	Approach	$\Delta\%$
Acceptance	19.42%	97.63%	402.63%
Rejection	80.58%	2.37%	-97.06%
Total	1,179	1,180	0.08%
Accepted	229	1,152	403.06%
Conflicts	950	981	3.26%
Actual Conflict %	80.58%	83.14%	3.18%
Returned	950	28	-97.05%
Aborted	0	0	0.00%
Redressed	0	953	-
# Average	37.564	38.735	3.12%
# Total	44,049	45,309	2.86%
# Accepted	8,760	44,152	404.02%
# Returned	35,289	1,157	-96.72%
# Aborted	0	0	0.00%
# Redressed	0	36,567	-

(b) Metrics

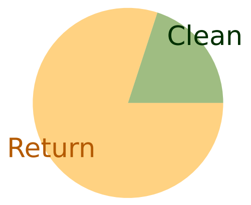
Table 5.9: Scenario 8 Details



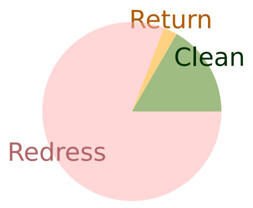
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.10: Scenario 8

Scenario 9

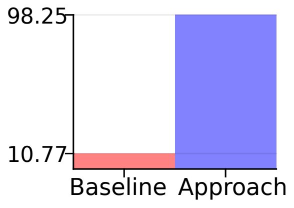
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	90%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

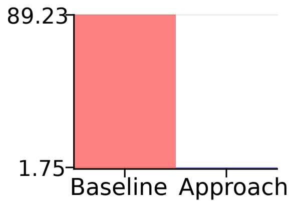
Metric	Baseline	Approach	$\Delta\%$
Acceptance	10.77%	98.25%	812.21%
Rejection	89.23%	1.75%	-98.04%
Total	1,142	1,143	0.09%
Accepted	123	1,123	813.01%
Conflicts	1,019	1,040	2.06%
Actual Conflict %	89.23%	90.99%	1.97%
Returned	1,019	20	-98.04%
Aborted	0	0	0.00%
Redressed	0	1020	-
# Average	37.781	38.324	1.44%
# Total	43,055	43,360	0.71%
# Accepted	4,795	42,571	787.82%
# Returned	38,260	789	-97.94%
# Aborted	0	0	0.00%
# Redressed	0	38,722	-

(b) Metrics

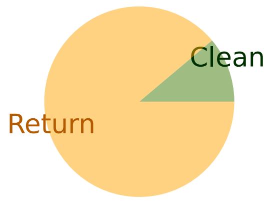
Table 5.10: Scenario 9 Details



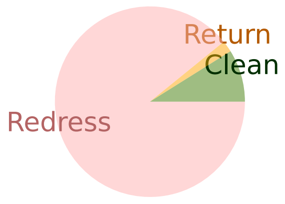
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.11: Scenario 9

Scenario 10

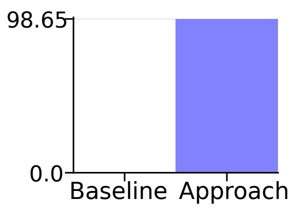
Parameter	Value
Initial Pool	500
Minimum Size	25
Maximum Size	50
Guarantee%	100%
PC%	0%
Delete%	0.5%
Create%	30%

(a) Input Configuration

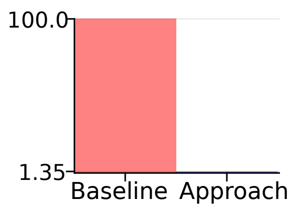
Metric	Baseline	Approach	$\Delta\%$
Acceptance	0.00%	98.65%	-
Rejection	100.00%	1.35%	-98.65%
Total	1,186	1,187	0.08%
Accepted	0	1,171	-
Conflicts	1,186	1,187	0.08%
Actual Conflict %	100.00%	100.00%	0.00%
Returned	1,186	16	-98.65%
Aborted	0	0	0.00%
Redressed	0	1,171	-
# Average	37.590	38.015	1.13%
# Total	44,357	44,511	0.35%
# Accepted	0	43,913	-
# Returned	44,357	598	-98.65%
# Aborted	0	0	0.00%
# Redressed	0	43,913	-

(b) Metrics

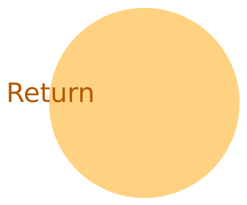
Table 5.11: Scenario 10 Details



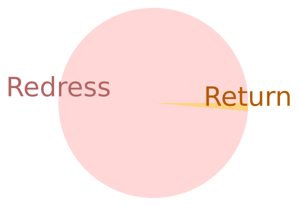
(a) Change Set Acceptance



(b) % Items Returned



(c) Relative Fate of Baseline Sets



(d) Relative Fate of Approach Sets

Fig. 5.12: Scenario 10

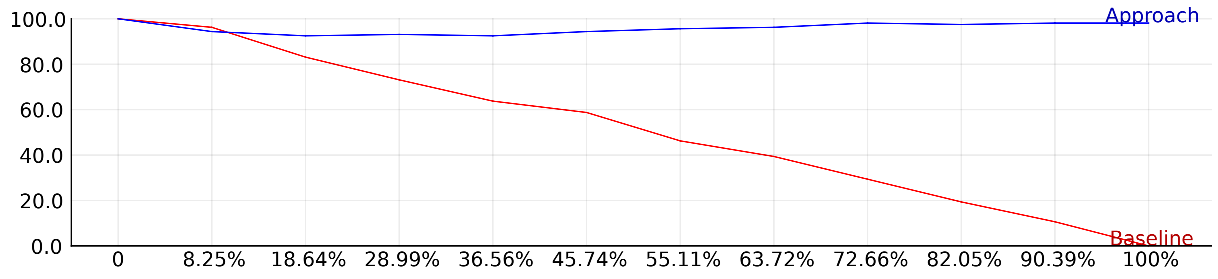


Fig. 5.13: Comparison of acceptance rates to conflict probability

The acceptance rates from the tests above were compiled into figure 5.13. The figure uses the median Conflict % with the associated Acceptance metric to illustrate more clearly how the approach fairs in scenarios where the expectation of conflicts is high. As the figure shows, the approach above can achieve near last write wins acceptance of change sets by relaxing the definition of coherence without compromising on strict consistency or sacrificing concurrency control.

5.2.4 Pessimistic Opt-in

In a situation where the data to be updated sees frequent modifications, it may be useful to halt concurrency temporarily while a critical body of work is accommodated. To test such a scenario, the following test was designed based on the provider-consumer pattern. The tests leveraged in the previous section are set to run on small data sets with 25 users concurrently participating. Another test is run, emulating a single user. The single user test is profiled for individual success rates and the worker pool from above, while offered for reference is disregarded as it is already known that once the lock is applied, all others immediately cease to be able to operate. Per the configuration in table 5.12a, the locker user makes two passes at the data. The first round is configured for 100% conflict guarantee and 100% chance to acquire a PC lock while using the RejectConcurrentEditsConflictResolver to guarantee the CS rejects completely. If the first round rejected properly, the user immediately submits a new CS, this time with 0% conflict guarantee and 0% chance to acquire a lock on failure and still uses the RejectConcurrentEditsConflictResolver. Four performance counters are used to capture the success rates of the locking process. Two count attempts, one before the first request, and one before the second. Two more count the successes of the requests. As the results in table 5.12a clearly demonstrate, this process was 100% effective is guaranteeing the locking process ensured success of the second write attempt.

Parameter	Value
Initial Pool	10
Minimum Size	10
Maximum Size	11
Guarantee% (worker)	0%
Guarantee% (locker)	100%, 0%
PC% (worker)	0%
PC% (locker)	100%, 0%
Delete%	0%
Create%	0%

(a) Input Configuration

Metric	Approach
Acceptance	3.40%
Rejection	95.24%
Total	62,397
Accepted	2,123
Conflicts	59,427
Actual Conflict %	95.24%
Returned	12,829
Aborted	46,985
Redressed	0
# Average	10
# Total	615,500
# Accepted	21,230
# Returned	128,290
# Aborted	46,598
# Redressed	0

(b) Metrics for 25 Workers

Metric	Count
First Round Attempts	3,954
First Round Successes	3,954
Second Round Attempts	3,954
Second Round Success	3,954
Comprehensive Success Rate	100%

(c) Metrics for Locker

Table 5.12: Pessimistic Lock Test Details

CHAPTER 6

CONCLUSION

The approach presented in this work leans on tried and true patterns of establishing optimistic concurrency control coupled with optional lock escalation to provide guarantees as to the determinate nature of conventionally conflicting transactions. A generic means of promoting any generic POCO (plain old common object) as a virtually conflict free data type within the confines of a custom tailored transaction control was presented. The composition of both features as a single solution presents opportunities in the world of off line data processing which are traditionally pigeonholed by constraints of coherence and difficult to manage. The small surface area, extensibility, unobtrusive and passive qualities of the merge framework provided make it suitable for a broad domain of semantic data processing applications.

6.1 Property Merge Enhancements

The source code in this project has taken into account the issue of boxing and unboxing of value types. In the .NET framework, value based types are passed by value on the stack, however, methods which accept arguments of type System.Object will incur a boxing operation when passed a value type. This boxing operation will wrap the value with a reference type object and place the object in the heap. Future references to the object will acquire the item from the heap and unbox it back to a value type where it will be copied into the stack at the memory location of the local variable using it. These operations, while small, can become very expensive cumulatively. One area of the property merge which has not fully resolved this issue is in the Rel.Merge.Strategies.MergeableAttribute's Merge method and its descendants.

The Merge method signature is a generic method so it will not incur boxing operations, however, there is no mechanism currently to get the properly types value of the argument without up-casting it to an object which incurs the boxing operations. Being that this is called on property merges, this should be addressed.

6.2 Data Access Layer Considerations

This project has wholly segregated the responsibility of data access out of its core functionality for portability. The DAL itself is very basic. Building a more elaborate DAL atop a RDBMS back-end may prove to have challenges, namely in the way of deadlock considerations. It works fine in this example implementation to place the PC lock in-line on the partition, but in a more complex system, it would be advantageous to port the PC lock to its own segment of the system to with specialized wrappers. Furthermore, the EF library's pattern of single command single entity works well for general use, but as stated previously makes for a very poor performing batch update system. Any DAL seeking to incorporate large change sets into its core functionality should invest effort up front in modeling the data domain and DAL to accommodate such behaviors.

APPENDIX A

CLIENT API DOCUMENTATION

`authenticationService`

service in module `app`

Description

Manages communication with the server regarding authentication information. This service is also responsible to handling offline user authentication.

Methods

`identify(username)`

Pings the server with a request for credential information as the server knows. The end point of this request is authenticated, so, if the browser submits an existing auth cookie back with this request, the result will be the authenticated username for the auth cookie used. This empowers session continuation between browser reloads.

Parameters

Param	Type	Details
username	**string ()**	The username to login under.

`jobService`

service in module `app`

Description

Exposes features oriented on Jobs.

Methods

`abandon(job)`

Drops all local changes and restores the job specified to an online state.

Parameters

Param	Type	Details
job	**Job ()**	The job to abandon offline changes from.

Returns

Q ()	A promise which, when resolved, signifies successful removal of all offline data for the job specified.

`checkin(job)`

Assembles a change set of offline changes for a job and submits them for processing at the server. If successful, the job specified is returned to an online state.

Parameters

Param	Type	Details
job	**Job ()**	The job to check in offline changes from.

Returns

Q ()	A promise which, when resolved, signifies successful completion of a check in operation and subsequent local cleanup of the offline store.

`checkout(job)`

Acquires a partition of data for offline processing.

Parameters

Param	Type	Details
job	**Job ()**	The job to check out.

Returns

Q () A promise which, when resolved, signifies successful completion of a check out operation in whole.

`getAll(remoteTruth)`

List all jobs available to the application.

Parameters

Param	Type	Details
remoteTruth *(optional)*	**boolean ()**	A flag to indicate how offline and online changes should be treated. If `true` is specified, then online results will be yielded with only flags indicating the offline status of the job. This is in contrast to returning offline results where the user may not see any changes which have been made. The `true` behavior is useful to render the project listing with information such as pessimistic locks applied since checkout. *(default: undefined)*

Returns

Q () A promise which, when resolved, yields the list of all jobs currently available to the application.

`getAssetAreas(job)`

Gets all known asset areas for the given job.

Parameters

Param	Type	Details
job	**Job ()**	The job to find asset areas for.

Returns

Q () A promise which, when resolved, yields an Array of distinct strings representing the known asset areas.

`getAssetByJobAndId(job)`

Gets all assets by the id and offline status of the given job.

Parameters

Param	Type	Details
job	**Job ()**	The job containing the assets to process.

Returns

Q () A promise which, when resolved, yields an Array of all assets currently available to the requested `Job`.

`getAssetByJobAndId(job)`

Gets an asset (online or offline) by the job status and assetId.

Parameters

Param	Type	Details
job	**Job ()**	The job containing the asset specified.

Returns

Q ()	A promise which, when resolved, yields the asset requested, or rejects on not found or error.

`getById(id)`

Get a specific job (online or offline) by id.

Parameters

Param	Type	Details
id	**number ()**	The identity of the job to acquire.

Returns

Q ()	A promise which, when resolved, yields the `Job` requested. Or rejects with an error or not found.

`isLocked(job)`

Interprets the locked status of a job.

Parameters

Param	Type	Details
job	**Job ()**	The job to inspect.

Returns

boolean ()	`true` if the `job` is locked; otherwise, `false`.

`lockedByMe(job, identity)`

Interprets the locked status of a job to determine if the identity specified holds a pessimistic lock on the given job.

Parameters

Param	Type	Details
job	**Job ()**	The job to inspect.

Param	Type	Details
identity	**Identity ()**	The identity of a user to test.

Returns

boolean () `true` if the `job` is locked by the specified `identity`; otherwise, `false`.

`lockedByOther(job, identity)`

Interprets the locked status of a job to determine if the identity specified does not hold the pessimistic lock on the given job.

Parameters

Param	Type	Details
job	**Job ()**	The job to inspect.
identity	**Identity ()**	The identity of a user to test.

Returns

boolean () `true` if the `job` is locked and not by the specified `identity`; otherwise, `false`.

API Documentation / app / LoginController

`LoginController`

service in module `app`

Description

Oversees the login process.

In this demonstration, the authentication is limited to only notifying the server what the active username is. We are not interested in a full blown user account or profile and passwords.

Methods

`login(username)`

Attempts to login through the authenticationService and close on success.

Parameters

Param	Type	Details
username	**string ()**	The username to login under.

`loginService`

service in module `app`

Description

Acquires credentials to submit for authentication.

Methods

`login([seed])`

Acquires credentials to submit for authentication.

This implementation uses Bootstrap to present a modal login dialog prompt.

Parameters

Param	Type	Details
[seed]	**object ()**	Provides seed values to the user credential acquisition process.

Returns

Q ()	A promise which resolves with authenticated user credentials.

API Documentation / app / offlineStore

`offlineStore`

service in module `app`

Description

Encapsulates the offline store and shadow copy functionality. When interpreting methods within the offline store:

- TOP refers to the original values placed in the store.
- BOTTOM refers to the modified values in the store.

In the store, unless specified otherwise, BOTTOM always supersedes TOP in read operations.

Methods

`get(type, id, suppressShadow)`

Gets an item from the store.

Parameters

Param	Type	Details
type	**string ()**	The type of object/store to pull from.
id	**string ()**	The identity of the item.
suppressShadow *(optional)*	**boolean ()**	`true` to suppress the shadow copy logic.

Returns

Q ()	A promise which, when resolved, yields an item from the store or rejects when not found.

`getAll(type, keyPath, suppressShadow)`

Gets all items from the store.

Parameters

Param	Type	Details
type	**string ()**	The type of object/store to pull from.
keyPath *(optional)*	**string ()**	The key property to resolve identity of items from.
suppressShadow *(optional)*	**boolean ()**	`true` to suppress the shadow copy logic.

Returns

Q ()	A promise which, when resolved, yields all items from the store.

`getByIndex(type, index, keyPath, filter, suppressShadow)`

Gets all items from the `type` store using a the specified `index` for query and which satisfy the given `filter`. Returns a promise wrapping the query of all items in the specified type store with a matching index. Items will be returned from both shadow copy and original values regions. Each item will exist only once, if it exists in the shadow copy region, the shadow will be returned. Otherwise, the original values are yielded.

Parameters

Param	Type	Details
type	**string ()**	The type of object/store to pull from.
index	**string ()**	The index name to crawl.
keyPath *(optional)*	**string ()**	The key property to resolve identity of items from.
filter *(optional)*	**function ()**	A callback to filter the results.
suppressShadow *(optional)*	**boolean ()**	`true` to suppress the shadow copy logic.

Returns

Q ()	A promise which, when resolved, yields all items from the store discovered by `index` passing the specified `filter` and found distinct by `keyPath`.

`purgeById(type, id, suppressShadow)`

Removes a specific item from the store.

Parameters

Param	Type	Details
type	**string ()**	The type of object/store to purge from.
id	**object ()**	The identity of the object.
suppressShadow *(optional)*	**boolean ()**	`true` to suppress the shadow copy logic.

Returns

Q ()	A promise which, when resolved, signifies completion of the purge.

`purgeByIndex(type, dbIndex, dbIndexValue, keyPath, suppressShadow)`

Removes all items from the store where the given `dbIndex` equals the specified `dbIndexValue`.

Parameters

Param	Type	Details
type	**string ()**	The type of object/store to purge from.

Returns

Q ()	A promise which, when resolved, signifies completion of the purge.

`put(type, items, suppressShadow)`

Insert or update one or more items in the store.

Parameters

Param	Type	Details
type	**string ()**	The type of object/store of the items.
items	**object ()** **array ()**	The items to be placed in the store.
suppressShadow *(optional)*	**boolean ()**	`true` to suppress the shadow copy logic.

Returns

Q ()	A promise which, when resolved, signifies the successful completion of the insert or update operation.

`query(type, callback, suppressShadow)`

Gets zero or more items from the store.

Parameters

Param	Type	Details
type	**string ()**	The type of object/store to pull from.
callback	**function ()**	A callback which receives a store and returns a {Q} of items from the store.
suppressShadow *(optional)*	**boolean ()**	`true` to suppress the shadow copy logic.

Returns

Q ()	A promise which, when resolved, yields all items from the store which satisfy the query.

`principal`

service in module `app`

Description

Manages the authenticated user credentials.

Methods

`identity(flush)`

Get the authenticated user credentials.

Parameters

Param	Type	Details
flush *(optional)*	**boolean ()**	Flags principal to drop cached credentials and acquire new credentials from the loginService. *(default: undefined)*

API Documentation / app / utils

`utils`

service in module `app`

Description

Provides common utility methods.

Methods

`coerceToInt(value)`

Converts a value to an integer.

Parameters

Param	Type	Details
value	* ()	The value to coerce to numeric.

Returns

number ()	The integer represented by value or 0.

`concatDistinct(rank, file, key)`

Concatenates file to rank, ensuring that only one item with an arbitrary key value exists in the resulting array. If items `A` and `B` have key value `x` the item elected into the resultant array is computed as follows:

- If `A` and `B` are both in `rank`, which ever appears last is taken.
- If `A` is in `rank` and `B` is in `file`, `A` is taken.
- If `A` and `B` are both in `file`, which ever appears first is taken.

Parameters

Param	Type	Details
rank	**array ()**	An array of values.
file	**array ()**	An array of values.
key	**string ()**	An optional keyPath to use in indexing the arrays. *(default: 'id')*

Returns

array ()	An array containing all elements resolved as distinct on the specified key.

`concatResults(pred, [keyPath])`

Wraps concatDistinct as a delegate for convenience in `Q` chaining.

Parameters

Param	Type	Details
pred	**array ()**	Predecessor query results. Will be used as the rank value in concatDistinct.
[keyPath]	**string ()**	Indexing property of both arrays.

Example

query.find().then(concatResults(previousQueryResults, 'id'));

`confirm(question)`

Presents an OK/Cancel dialog prompt.

Parameters

Param	Type	Details
question	**string ()**	The question to prompt with.

Returns

Q ()	A promise which resolves when OK is chosen, or rejects if canceled.

`findById(array, keyValue, keyPath)`

Finds an object in an array by an arbitrary index value.

Parameters

Param	Type	Details
array	**array ()**	The Array object to search.
keyValue	**string ()**	The identity value sought.
keyPath	**string ()**	The property containing the key value. *(default: 'id')*

Returns

object ()	The object identified by the given id.

`fixObjGraph(graph, indices)`

Restores a deserialized object graph which used the $id, $ref, and $values attribute convention to serialize cycles.

Parameters

Param	Type	Details
graph	**object ()**	The root of the deserialized graph.

Param	Type	Details
indices	**object ()**	A pre-seeded index of $refs located in graph *(default: {})*

Returns

object ()	The corrected object graph.

`indexArray(array, key)`

Indexes an array on an arbitrary property.

Parameters

Param	Type	Details
array	**array ()**	The array to index.
key	**string ()**	The property containing the index value. *(default: 'id')*

Returns

object ()	An index object of the array.

APPENDIX B

CLIENT CODE

Core client code presented below.

B.1 Application Space

```
(function () {
    'use strict';


    /**
     * @ngdoc overview
     * @name app
     * @description
     *
     * Proof of Concept
     */
    angular.module('app', [
        'ngAnimate',
        'ngSanitize',

        'ui.router',
        'ui.bootstrap',

        'toastr',
        'angularMoment',
        'indexedDB'
    ]);
    angular.module('app').run(function ($rootScope) {
        $rootScope.$on("$stateChangeError", console.log.bind(console));
    });



})();
```

Listing B.1: app.js

```
(function () {
    'use strict';
    angular.module('app').config(function (toastrConfig) {
        angular.extend(toastrConfig, {
            positionClass: 'toast-bottom-center'
        });
    });


    angular.module('app').config(function ($logProvider) {
        $logProvider.debugEnabled(true);
    });


    angular.module('app').filter('numeric', function() {
        return function(input) {
            return parseInt(input, 10);
        };
    });
})();
```

Listing B.2: config.js

```
(function () {
    'use strict';

    var shadow = '_shadow',
        isShadow = new RegExp(shadow + '$');


    angular.module('app').config(function ($indexedDBProvider) {
        $indexedDBProvider
            .connection('thesis')
            .upgradeDatabase(1, function (event, db, tx) {
                createAndShadowCopy(db, 'job', { keyPath: 'id' }, [
                    ['name_idx', 'name', { unique: false }]
                ]);
                createAndShadowCopy(db, 'asset', { keyPath: 'id' }, [
                                    ['jobId, serviceArea', ['jobId', 'serviceArea'], {
                                        unique: false }],
                                    ['jobId', 'jobId', { unique: false }]
                ]);
            });


        function createAndShadowCopy(db, storeName, optionalParams, indices) {
            var objectStore = db.createObjectStore(storeName, optionalParams);
            for (var i = 0; i < indices.length; i += 1) {
                objectStore.createIndex.apply(objectStore, indices[i]);
            }

            if (!isShadow.test(storeName)) {
                createAndShadowCopy(db, storeName + shadow, optionalParams, indices);
            }
        }
```

```
});


/**
 * @ngdoc object
 * @name app.offlineStore
 * @description
 *
 * Encapsulates the offline store and shadow copy functionality.
 * When interpreting methods within the offline store:
 * * TOP refers to the original values placed in the store.
 * * BOTTOM refers to the modified values in the store.
 *
 * In the store, unless specified otherwise, BOTTOM always
 * supersedes TOP in read operations.
 */
angular.module('app').factory('offlineStore', offlineStore);

function offlineStore($q, $log, $indexedDB, utils) {
    var enabled = checkEnabled(),
        disabled = $q.reject('offline storage is unavailable'),
        svc = {
            get: get,
            query: query,
            getByIndex: getByIndex,
            put: put,
            getAll: getAll,
            track: track,
            purge: purgeByIndex,
            purgeById: purgeById
        };

    return svc;


    /**
     * @ngdoc function
     * @name app.offlineStore#get
     * @methodOf app.offlineStore
     * @description
     *
     * Gets an item from the store.
     *
     * @param {string} type The type of object/store to pull from.
     * @param {string} id The identity of the item.
     * @param {boolean=} suppressShadow `true` to suppress the
     * shadow copy logic.
     * @returns {Q} A promise which, when resolved, yields an item
     * from the store or rejects when not found.
     */
    function get(type, id, suppressShadow) {
        if (!enabled) { return disabled; }
        var promise =
            suppressShadow === true ?
            $q.reject() :
            get(type + shadow, id, true);
        return promise
            .then(null, function () {
                return openStore(type, _get(id));
            });
    }


    /**
     * @ngdoc function
     * @name app.offlineStore#query
     * @methodOf app.offlineStore
     * @description
     *
     * Gets zero or more items from the store.
     *
     * @param {string} type The type of object/store to pull from.
     * @param {function} callback A callback which receives a store
     * and returns a {Q} of items from the store.
     * @param {boolean=} suppressShadow ‘true‘ to suppress the
     * shadow copy logic.
     * @returns {Q} A promise which, when resolved, yields all
     * items from the store which satisfy the query.
     */
    function query(type, callback, suppressShadow) {
        if (!enabled) { return disabled; }
        var promise =
            suppressShadow === true ?
            $q.when([]) :
            getAll(type + shadow, callback, true);

        return promise
            .then(function (pred) {
                return openStore(type, callback)
                    .then(function (a) { return pred.concat(a); });
            });
    }

    /**
     * @ngdoc function
     * @name app.offlineStore#getAll
     * @methodOf app.offlineStore
     * @description
     *
     * Gets all items from the store.
     *
     * @param {string} type The type of object/store to pull from.
     * @param {string=} keyPath The key property to resolve
     * identity of items from.
     * @param {boolean=} suppressShadow ‘true‘ to suppress the
     * shadow copy logic.
     * @returns {Q} A promise which, when resolved, yields all
     * items from the store.
     */
    function getAll(type, keyPath, suppressShadow) {
        if (!enabled) { return disabled; }
        var promise =
            suppressShadow === true ?
            $q.when([]) :
            getAll(type + shadow, keyPath, true);

        return promise
            .then(function (pred) {
                return openStore(type, function (store) {
                    return store.getAll()
                        .then(utils.concatResults(pred, keyPath));
                });
            });
    }



    /**
     * @ngdoc function
     * @name app.offlineStore#getByIndex
     * @methodOf app.offlineStore
     * @description
     *
     * Gets all items from the ‘type‘ store using a the specified
     * ‘index‘ for query and which satisfy the given ‘filter‘.
     * Returns a promise wrapping the query of all items in the
     * specified type store with a matching index. Items will be
     * returned from both shadow copy and original values regions.
     * Each item will exist only once, if it exists in the shadow
     * copy region, the shadow will be returned.  Otherwise, the
     * original values are yielded.
     *
     * @param {string} type The type of object/store to pull from.
     * @param {string} index The index name to crawl.
     * @param {string=} keyPath The key property to resolve
     * identity of items from.
     * @param {function=} filter A callback to filter the results.
     * @param {boolean=} suppressShadow ‘true‘ to suppress the
     * shadow copy logic.
     * @returns {Q} A promise which, when resolved, yields all
     * items from the store discovered by ‘index‘ passing the
     * specified ‘filter‘ and found distinct by ‘keyPath‘.
     */
    function getByIndex(type, index, keyPath, filter, suppressShadow) {
        if (!enabled) { return disabled; }
        var promise =
            suppressShadow === true ?
            $q.when([]) :
            getByIndex(type + shadow, index, keyPath, filter, true);
        $log.debug('getByIndex', arguments);
        return promise
            .then(function (pred) {
                return openStore(type, _getByIndex(index, _filter(filter)))
                    .then(function (p) {
                        return utils.concatResults(pred, keyPath)(p);
                    });
            });
```

```
}



/**
 * @ngdoc function
 * @name app.offlineStore#purgeById
 * @methodOf app.offlineStore
 * @description
 *
 * Removes a specific item from the store.
 *
 * @param {string} type The type of object/store to purge from.
 * @param {object} id The identity of the object.
 * @param {boolean=} suppressShadow `true` to suppress the
 * shadow copy logic.
 * @returns {Q} A promise which, when resolved, signifies
 * completion of the purge.
 */
function purgeById(type, id, suppressShadow) {
    if (!enabled) { return disabled; }
    var promise =
        suppressShadow === true ?
        $q.when() :
        purgeById(type + shadow, id, true);

    return promise
        .finally(openAndDeleteStoreById(type, id));
    //.catch(angular.noop);
}

/**
 * @ngdoc function
 * @name app.offlineStore#purgeByIndex
 * @methodOf app.offlineStore
 * @description
 *
 * Removes all items from the store where the given `dbIndex`
 * equals the specified `dbIndexValue`.
 *
 * @param {string} type The type of object/store to purge from.
 * @param {string} dbIndex The index to crawl.
 * @param {*} dbIndexValue The value to find.
 * @param {string} [keyPath='id'] The identity property of the
 * items to be deleted.
 * @param {boolean=} suppressShadow `true` to suppress the
 * shadow copy logic.
 * @returns {Q} A promise which, when resolved, signifies
 * completion of the purge.
 */
function purgeByIndex(type, dbIndex, dbIndexValue, keyPath, suppressShadow) {
    if (!enabled) { return disabled; }
    var promise =
        suppressShadow === true ?
        $q.when() :
        purgeByIndex(type + shadow, dbIndex, dbIndexValue, keyPath, true);
    return promise.finally(openAndDeleteStore(type, dbIndex, dbIndexValue,
        keyPath));
}


/**
 * @ngdoc function
 * @name app.offlineStore#put
 * @methodOf app.offlineStore
 * @description
 *
 * Insert or update one or more items in the store.
 *
 * @param {string} type The type of object/store of the items.
 * @param {object|array} items The items to be placed in the store.
 * @param {boolean=} suppressShadow ‘true‘ to suppress the
 * shadow copy logic.
 * @returns {Q} A promise which, when resolved, signifies the
 * successful completion of the insert or update operation.
 */
function put(type, items, suppressShadow) {
    if (!enabled) { return disabled; }
    if (suppressShadow !== true) {
        type = type + shadow;
    }

    return openStore(type, function (store) {
        return store.upsert(items);
    });
}



function openAndDeleteStore(storeName, dbIndex, dbIndexValue, keyPath) {
    return function () {
        return openStore(storeName, function (store) {
            var find = store.query().$eq(dbIndexValue).$index(dbIndex);
            return store.eachWhere(find).then(function (a) { return
                deleteThese(a, keyPath, store); });
        });
    };
}



function openAndDeleteStoreById(storeName, itemKey) {
    return function () {
        return openStore(storeName, function (store) {
            return store.delete(itemKey);
        });
    };
}


function deleteThese(a, keyPath, store) {
    var keys = a.map(function (e) { return e[keyPath]; });
    return $q.all(keys.map(store.delete, store));
}


function track(type, items) {
    return openStore(type, function (store) {
        return store.upsert(items);
    });
}


function _get(id) { return function (store) { return store.find(id); }; }


function _select(a) { return a; }


function _filter(filterFn) {
    return function (a) {
        var results = a.filter(filterFn);
        $log.debug('_filter %o -> %o', arguments, results);
        return results;
    };
}


function _getByIndex(index, filter) {
    return function (store) {
        return store.eachBy(index).then(filter);
    };
}

function logError(e) {
    $log.log(e, e);
    return $q.reject(e);
}





function openStore(name, callback) {
    var promise;
    if (angular.isFunction(name)) {
        callback = name;
        name = 'job';
    } else {
        name = name || 'job';
    }

    try {
        promise = $indexedDB.openStore(name, callback);
    } catch (e) {
        return $q.reject(e);
```

```
            }

            return promise;
        }

        function checkEnabled() {
            return true;//TODO
        }
    }
})();
```

Listing B.3: common/indexeddb.config.js

```
(function () {
    'use strict';

    angular.module('app').factory('utils', utils);


    /**
     * @ngdoc object
     * @name app.utils
     * @description
     *
     * Provides common utility methods.
     */
    function utils($uibModal) {
        var service = {
            findById: findById,
            fixObjGraph: fixObjGraph,
            confirm: confirm,
            concatDistinct: concatDistinct,
            concatResults: concatResults,
            coerceToInt: coerceToInt,
            indexArray: indexArray
        };
        return service;





        /**
         * @ngdoc function
         * @name app.utils#coerceToInt
         * @methodOf app.utils
         * @description
         *
         * Converts a value to an integer.
         *
         * @param {*} value The value to coerce to numeric.
         * @returns {number} The integer represented by value or 0.
         */
        function coerceToInt(value) {
```

```
    return ~~value;
}




/**
 * @ngdoc function
 * @name app.utils#findById
 * @methodOf app.utils
 * @description
 *
 * Finds an object in an array by an arbitrary index value.
 *
 * @param {array} array The Array object to search.
 * @param {string} keyValue The identity value sought.
 * @param {string} [keyPath='id'] The property containing the key value.
 * @returns {object} The object identified by the given id.
 */
function findById(array, id, keyPath) {
    keyPath = keyPath || 'id';
    for (var i = array.length; i--;) {
        if (array[i][keyPath] === id) {
            return array[i];
        }
    }
}




/**
 * @ngdoc function
 * @name app:utils#indexArray
 * @methodOf app.utils
 * @description
 *
 * Indexes an array on an arbitrary property.
 *
 * @param {array} array The array to index.
 * @param {string} [key='id'] The property containing the index value.
 * @returns {object} An index object of the array.
 */
function indexArray(array, key) {
    return array.reduce(function (p, c, i) {
        p[c[key]] = c;
        return p;
    }, {});
}



/**
 * @ngdoc function
 * @name app.utils#concatDistinct
 * @methodOf app.utils
 * @description
 *
 * Concatenates file to rank, ensuring that only one item with
 * an arbitrary key value exists in the resulting array.  If
 * items ‘A‘ and ‘B‘ have key value ‘x‘ the item elected into
 * the resultant array is computed as follows:
 *
 * * If ‘A‘ and ‘B‘ are both in ‘rank‘, which ever appears last
 * is taken.
 * * If ‘A‘ is in ‘rank‘ and ‘B‘ is in ‘file‘, ‘A‘ is taken.
 * * If ‘A‘ and ‘B‘ are both in ‘file‘, which ever appears
 * first is taken.
 *
 * @param {array} rank An array of values.
 * @param {array} file An array of values.
 * @param {string} [key='id'] An optional keyPath to use in
 * indexing the arrays.
 * @returns {array} An array containing all elements resolved
 * as distinct on the specified key.
 */
function concatDistinct(rank, file, key) {
    var index = indexArray(rank, key);

    return rank
        .concat(file.filter(function (e, i, a) {
            return !index.hasOwnProperty(e[key]);
        }));
}



/**
 * @ngdoc function
 * @name app.utils#concatResults
 * @methodOf app.utils
 * @description
 *
 * Wraps {@link app.utils#methods_concatDistinct concatDistinct}
 * as a delegate for convenience in ‘Q‘ chaining.
 *
 * @example query.find().then(concatResults(previousQueryResults, 'id'));
 *
 * @param {array} pred Predecessor query results. Will be used
 * as the rank value in concatDistinct.
 * @param {string} [keyPath] Indexing property of both arrays.
 */
function concatResults(pred, keyPath) {
    return function (r) {
        return concatDistinct(asArray(pred), asArray(r), keyPath);
    };
}
function asArray(e) { return angular.isArray(e) ? e : []; }
```

```
/**
 * @ngdoc function
 * @name app.utils#confirm
 * @methodOf app.utils
 * @description
 *
 * Presents an OK/Cancel dialog prompt.
 *
 * @param {string} question The question to prompt with.
 * @returns {Q} A promise which resolves when OK is chosen, or
 * rejects if canceled.
 */
function confirm(question) {
    var inst = $uibModal.open({
        animation: true,
        template: '<div class="modal-header">' +
                  '   <h2>Confirm</h2>' +
                  '</div>' +
                  '<div class="modal-body">' +
                  '    <p>' + question + '</p>' +
                  '</div>' +
                  '<div class="modal-footer">' +
                  '    <div>' +
                  '        <button data-ng-click="vm.close()" class="btn btn-
                      default">OK</button>' +
                  '        <button data-ng-click="vm.dismiss()" class="btn btn-
                      default">Cancel</button>' +
                  '    </div>' +
                  '</div>',
        controllerAs: 'vm',
        controller: function (close, dismiss) {
            this.close = close;
            this.dismiss = dismiss;
        },
        size: 'sm',
        backdrop: 'static',
        resolve: {
            close: function () {
                return function () {
                    return inst.close.apply(inst, arguments);
                };
            },
            dismiss: function () {
                return function () {
                    return inst.dismiss.apply(inst, arguments);
                };
            }
        }
    });

    return inst.result;
}
```

```
        /**
         * @ngdoc function
         * @name app.utils#fixObjGraph
         * @methodOf app.utils
         * @description
         *
         * Restores a deserialized object graph which used the $id,
         * $ref, and $values attribute convention to serialize cycles.
         *
         * @param {object} graph The root of the deserialized graph.
         * @param {object} [indices={}] A pre-seeded index of $refs
         * located in graph
         * @returns {object} The corrected object graph.
         */
        function fixObjGraph(graph, indices) {
            indices = indices || {};

            if (!graph || typeof graph === 'string') { return graph; }

            if (graph.hasOwnProperty('$ref')) {
                return indices[graph.$ref];
            }

            if (graph.hasOwnProperty('$id')) {
                if (graph.hasOwnProperty('$values')) {
                    graph = indices[graph.$id] = graph.$values;
                } else {
                    indices[graph.$id] = graph;
                    delete graph.$id;
                }
            }

            for (var member in graph) {
                if (graph.hasOwnProperty(member)) {
                    graph[member] = fixObjGraph(graph[member], indices);
                }
            }

            return graph;
        }
    }
})();
```

Listing B.4: common/utils.js

B.2 Job Facilities

```
(function () {
    'use strict';
    var componentId = 'CheckoutController';
    angular.module('app').controller(componentId, CheckoutController);


    function CheckoutController(action, job, identity, $state, utils, jobService,
        toastr) {
        var vm = angular.extend(this, job, {
            error: null,

            locked: jobService.isLocked(job),
            lockedByOther: jobService.lockedByOther(job, identity),
            lockedByMe: jobService.lockedByMe(job, identity)
        });



        assertJobStatusAndAction(vm, action);


        function goHome() {
            $state.go('job.details');
        }

        function reconcile() {
            // a state yet to be defined
            goHome();
        }



        function abandon() {
            return utils.confirm('One last time... Are you sure you want to do this?')
                .then(
                function () {
                    return jobService.abandon(job)
                            .then(function () {
                                toastr.info('Job wiped from offline storage.', job.
                                    name);
                                goHome();
                            });
                }, angular.noop);
        }



        function checkout() {
            if (vm.lockedByOther) {
                utils
                    .confirm('This job is currently locked by ' + job.lockedBy +
'. You have an increased chance of conflicts if you continue.  Click OK to accept the
    risk and checkout, or Cancel to cancel.')
                    .then(function () { proceedWithCheckout(job); }, angular.noop);
            } else {
                proceedWithCheckout();
            }
        }
```

```
function proceedWithCheckout() {
    jobService
        .checkout(job)
        .then(function () {
            toastr.info('Job checked out locally', job.name);
            goHome();
        }, function (e) {
            if (/locked by user/.test(e.exceptionMessage)) {
                toastr.warning(e.exceptionMessage, job.name);
            } else {
                toastr.error('An error was encountered while attempting to
                    check out this job.', job.name);
            }
        });
}


function checkin() {
    jobService
        .checkin(job)
        .then(function (_) {
            toastr.success("Your changes have been successfully checked in.");
            goHome();
        }, function (e) {
            if (e.staus === 400) {
                if (/locked by user/.test(e.data.exceptionMessage)) {
                    toastr.error(e.data.exceptionMessage, job.name);
                }
                else if (e.message === 'conflict') {
                    return utils.confirm('One or more of your changes
                        conflicts with changes already accepted at the server.
                          Please reconcile your changes and resubmit.')
                        .then(reconcile, goHome);
                }
            } else {
                toastr.error('There was an error submitting your changes.', '
                    Unable to check in');
            }
        });
}


function assertJobStatusAndAction(vm, action) {

    if (action === 'checkin') {
        vm.checkin = checkin;
        if (!job.offline) {
            toastr.info("Job is not checked out.", job.name);
            goHome();
        }
    } else if (action === 'checkout') {
        vm.checkout = checkout;
        if (job.offline) {
```

```
                    toastr.info("Job is already checked out.", job.name);
                    goHome();
                }
            } else if (action === 'abandon') {
                vm.abandon = abandon;
                if (!job.offline) {
                    toastr.info("Job is not checked out.", job.name);
                    goHome();
                }
            } else {
                throw 'invalid action';
            }
        }
    }
})();
```

Listing B.5: jobs/job.checkout.js

```
(function () {
    'use strict';

    var componentId = 'jobService';

    angular.module('app').factory('jobService', jobService);

    function jobService($log, $q, $http, offlineStore, utils, toastr, principal) {
        var __reportedOffline = false,
            __reportedOnline = false,
            /**
             * @ngdoc object
             * @name app.jobService
             * @description
             *
             * Exposes features oriented on Jobs.
             */
            service = {
                abandon: abandon,
                checkin: checkin,
                checkout: checkout,
                getAll: getAll,
                getById: getById,
                isLocked: isLocked,

                lockedByMe: lockedByMe,
                lockedByOther: lockedByOther,

                /*** Asset Utilities ***/
                getAssetAreas: getAssetAreas,
                getAssetByJobAndId: getAssetByJobAndId,
                getAssets: getAssets,
                save: save
            };

        return service;
```

```
/**
 * @ngdoc function
 * @name app.jobService#abandon
 * @methodOf app.jobService
 * @description
 *
 * Drops all local changes and restores the job specified to an
 * online state.
 *
 * @param {Job} job The job to abandon offline changes from.
 * @returns {Q} A promise which, when resolved, signifies
 * successful removal of all offline data for the job specified.
 */
function abandon(job) {
    if (!job) { return $q.reject('Job is required'); }
    if (!job.offline) { return $q.reject('job is offline'); }

    // job was available locally so begin deleting all associated
    // data, working our way back up to delete the job last. This
    // way if there are any issues encountered, we don't end up with
    // a job partially on- and registered as fully off-line.
    return offlineStore.purge('asset', 'jobId', job.id, 'id')
        .then(function () { return offlineStore.purgeById('job', job.id); })
        .finally(clearOfflineStatus(job))
        .catch(handleOfflineError);
}

/**
 * @ngdoc function
 * @name app.jobService#checkin
 * @methodOf app.jobService
 * @description
 *
 * Assembles a change set of offline changes for a job and
 * submits them for processing at the server.  If successful,
 * the job specified is returned to an online state.
 *
 *
 *
 * @param {Job} job The job to check in offline changes from.
 * @returns {Q} A promise which, when resolved, signifies
 * successful completion of a check in operation and subsequent
 * local cleanup of the offline store.
 */
function checkin(job) {
    if (!job) { return $q.reject('Job is required'); }

    return compileChangeset(job)
        .then(function (changeSet) { return { partitionId: job.id,
            claimPartition: null, changeSet: changeSet }; })
        .then(sendChanges)
        .then(function (upd) {
            // successful check in so abandon local changes as
            // this job is now back online.
            return abandon(job);
        }, function (j) {
            job.reconcile = true;
            offlineStore.put('job', job, true);
            return $q.reject(job);
        });
}

/**
 * @ngdoc function
 * @name app.jobService#checkout
 * @methodOf app.jobService
 * @description
 *
 * Acquires a partition of data for offline processing.
 *
 * @param {Job} job The job to check out.
 * @returns {Q} A promise which, when resolved, signifies
 * successful completion of a check out operation in whole.
 */
function checkout(job) {
    if (!job) { return $q.reject('Job is required'); }

    var j = angular.copy(job);
    j.offline = true;
    j.offlineSince = new Date();

    return getAssets(job)
            .then(function (r) {
                return offlineStore.track('asset', r)
                    .then(function () {
                        offlineStore.track('job', j);
                    }, rethrowOfflineError)
                    .then(function () {
                        return angular.extend(job, j);
                    }, rethrowOfflineError);
            }, rethrowOfflineError);
}

/**
 * @ngdoc function
 * @name app.jobService#getAll
 * @methodOf app.jobService
 * @description
 *
 * List all jobs available to the application.
 *
 * @param {boolean=} [remoteTruth=undefined] A flag to indicate
 * how offline and online changes should be treated. If `true`
 * is specified, then online results will be yielded with only
 * flags indicating the offline status of the job.  This is in
 * contrast to returning offline results where the user may not
 * see any changes which have been made. The `true` behavior is
 * useful to render the project listing with information such
 * as pessimistic locks applied since checkout.
 *
 * @returns {Q} A promise which, when resolved, yields the list
 * of all jobs currently available to the application.
 */
function getAll(remoteTruth) {
    var offlinePromise,
        onlinePromise;

    offlinePromise = offlineStore.getAll('job')
        .then(null, handleOfflineError);

    onlinePromise = $http.get('/api/jobs')
        .then(selectAndFixResponseData, handleOnlineError);

    return $q.all([offlinePromise, onlinePromise])
        .then(function (results) {
            return mergeResults(results[0], results[1], remoteTruth === true);
        });
}

/**
 * @ngdoc function
 * @name app.jobService#getById
 * @methodOf app.jobService
 * @description
 *
 * Get a specific job (online or offline) by id.
 *
 * @param {number} id The identity of the job to acquire.
 * @returns {Q} A promise which, when resolved, yields the `Job`
 * requested. Or rejects with an error or not found.
 */
function getById(id) {
    if (!id) { return $q.reject('Job ID is required'); }

    return offlineStore.get('job', id)
        .catch(function (e) { // cache miss
            return $http.get('/api/jobs/' + id)
                .then(selectAndFixResponseData);
        });
}



/**
 * @ngdoc function
 * @name app.jobService#isLocked
 * @methodOf app.jobService
 * @description
 *
 * Interprets the locked status of a job.
 *
 * @param {Job} job The job to inspect.
 * @returns {boolean} `true` if the `job` is locked; otherwise,
 * `false`.
 */
function isLocked(job) {
    if (!job) { return $q.reject('Job is required'); }
    return !!job.lockedBy;
}

/**
 * @ngdoc function
 * @name app.jobService#lockedByMe
 * @methodOf app.jobService
 * @description
 *
 * Interprets the locked status of a job to determine if the
 * identity specified holds a pessimistic lock on the given job.
 *
 * @param {Job} job The job to inspect.
 * @param {Identity} identity The identity of a user to test.
 * @returns {boolean} `true` if the `job` is locked by the
 * specified `identity`; otherwise, `false`.
 */
function lockedByMe(job, identity) {
    return !!job.lockedBy && job.lockedBy === identity.username;
}

/**
 * @ngdoc function
 * @name app.jobService#lockedByOther
 * @methodOf app.jobService
 * @description
 *
 * Interprets the locked status of a job to determine if the
 * identity specified does not hold the pessimistic lock on the
 * given job.
 *
 * @param {Job} job The job to inspect.
 * @param {Identity} identity The identity of a user to test.
 * @returns {boolean} `true` if the `job` is locked and not by
 * the specified `identity`; otherwise, `false`.
 */
function lockedByOther(job, identity) {
    if (angular.isString(job)) {
        job = { lockedBy: job };
    }

    if (angular.isString(identity)) {
        identity = { username: identity };
    }
    return !!job.lockedBy && job.lockedBy !== identity.username;
}

/**
 * @ngdoc function
 * @name app.jobService#getAssetAreas
 * @methodOf app.jobService
 * @description
 *
 * Gets all known asset areas for the given job.
 *
 * @param {Job} job The job to find asset areas for.
 * @returns {Q} A promise which, when resolved, yields an Array
 * of distinct strings representing the known asset areas.
 */
function getAssetAreas(job) {
    if (job.offline) {
        return offlineStore.query('asset', function (store) {
            var query = store
                .query()
                .$index('jobId, serviceArea')
                .$asc(true);
            return store.eachWhere(query).then(function (a) {
                return a.filter(function (i) { return i.jobId === job.id; });
            });
        }).then(function (a) {
            var r = [];
            a.reduce(function (p, c, i) {
                var v = c.serviceArea;
                if (!p.hasOwnProperty(v)) {
                    r.push(c.serviceArea);
                    p[v] = c;
                }
                return p;
            }, {});

            return r;
        });
    }

    return $http({
        url: '/api/assets/areas',
        params: { jobId: job.id },
        paramSerializer: '$httpParamSerializerJQLike'
    })
    .then(selectAndFixResponseData);
}



/**
 * @ngdoc function
 * @name app.jobService#getAssetByJobAndId
 * @methodOf app.jobService
 * @description
 *
 * Gets an asset (online or offline) by the job status and
 * assetId.
 *
 * @param {Job} job The job containing the asset specified.
 * @returns {Q} A promise which, when resolved, yields the
 * asset requested, or rejects on not found or error.
 */
function getAssetByJobAndId(job, assetId) {
    if (job.offline) {
        return offlineStore.get('asset', assetId);
    }
    return $http({
        url: '/api/asset',
        params: { jobId: job.id, assetId: assetId },
        paramSerializer: '$httpParamSerializerJQLike'
    }).then(selectAndFixResponseData, function (e) {
        $log.error("failed getting asset " + assetId + " in job " + job.id, e)
            ;
    });
}

/**
 * @ngdoc function
 * @name app.jobService#getAssetByJobAndId
 * @methodOf app.jobService
 * @description
 *
 * Gets all assets by the id and offline status of the given
 * job.
 *
 * @param {Job} job The job containing the assets to process.
 * @returns {Q} A promise which, when resolved, yields an Array
 * of all assets currently available to the requested `Job`.
 */
function getAssets(job, areaFilter, suppressShadow) {
    if (job.offline) {
        if (areaFilter) {
            return offlineStore.getByIndex('asset', 'jobId, serviceArea', 'id'
                , function (a) {
                return job.id === a.jobId && a.serviceArea === areaFilter;
            });
        } else {
            return offlineStore.getAll('asset', suppressShadow);
        }
    }
    return $http({
        url: '/api/assets',
        params: { jobId: job.id, area: areaFilter },
        paramSerializer: '$httpParamSerializerJQLike'
    })
    .then(selectAndFixResponseData);
}

function save(job, asset) {
    if (job.offline) {
        return offlineStore.put('asset', asset)
        .then(function () { return asset; }, function (e) {
            $log.error('error while saving asset.', e);
            return $q.reject(e);
        });
    }

    return $http.put('/api/asset', asset)
        .then(selectAndFixResponseData, rethrowOnlineError);
}
```

```
function handleOfflineError(e) {
    if (!__reportedOffline && e && angular.isString(e) && /unavailable/.test(e
        )) {
        toastr.warning('Offline storage is unavailable.');
        __reportedOffline = true;
    }
    $log.error('jobService offline error:', e);
}
function rethrowOfflineError(e) {
    handleOfflineError(e);
    return $q.reject(e);
}
function rethrowOnlineError(e) {
    handleOnlineError(e);
    return $q.reject(e);
}
function handleOnlineError(e) {
    if (!__reportedOnline && e && e.status === -1) {
        toastr.warning('Online storage is unavailable.');
        __reportedOnline = true;
    }
    if (e.status === 400) {
        toastr.warning(e.data.message, e.statusText);
        return e.data;
    }
    $log.error('jobService online error:', e);
}
```

```
function reconcile(changeset) {
    return changeset;
}

function pickCurrentFromChangeSetItem(csi) {
    return csi.afim;
}

function handleChangeSetError(e) {
    if (e.status === 400) {
        $q.reject("Changeset is invalid. " + JSON.stringify(
            selectAndFixResponseData(e)));
    } else if (e.status === 409) {
        toastr.warning("Conflicts were detected.  Please review your changes
            and re-submit",'Check in');
        var changeset = selectAndFixResponseData(e);
        var assets = changeset.assets.map(pickCurrentFromChangeSetItem);
        return offlineStore
            .put('asset', assets, true)
            .then(function () {
                changeset.job.reconcile = true;
                return $q.reject(offlineStore.track(changeset.job));
            });
    } else {
        return $q.reject(e);
    }
}

function createChangeItems(a) {
    return a[1].map(join(a[0]));
}
function join(inner) {
    return function (outer) {
        var bfim = utils.findById(inner, outer.id);
        var item = { bfim: bfim, afim: outer, action: 'update' };
        $log.debug('submitting change item ', item);
        return item;
    };
}
function filterToJob(job) {
    return function (a) {
        return a.filter(function (b) { return b.jobId === job.id; });
    };
}
function compileChangeset(job) {
    return $q.all([
        offlineStore.getAll('asset', 'id', true).then(filterToJob(job)),
        offlineStore.getAll('asset_shadow', 'id', true).then(filterToJob(job))
    ])

        //offlineStore.getByIndex('asset', 'jobId', function (a) { return a.
            jobId === job.id; }, true),
        //offlineStore.getByIndex('asset_shadow', 'jobId', function (a) {
            return a.jobId === job.id; })])
        .then(createChangeItems)
        .then(function (changes) {
            changes.prototype = Array.prototype;
            return { assets: changes };
        });
}

function sendChanges(changeset) {
    return $http.post('/api/offline/checkin', changeset)
        .then(selectAndFixResponseData, handleChangeSetError);
}



function clearOfflineStatus(job) {
    return function () {

        delete job.offline;
        delete job.offlineSince;
        delete job.reconcile;
        return job;
    };
}
```

```
        // { offlineError: e, onlineError: e, offline: data, online: data }
        function mergeResults(offline, online, favorOnline) {
            if (!angular.isArray(offline) && !angular.isArray(online)) {
                $log.error('Unable to access either online or offline data');
                return $q.reject('No offline data available and unable to access
                    online store');
            }

            offline = angular.isArray(offline) ? offline : [];
            online = angular.isArray(online) ? online : [];
            offline.unshift.apply(offline, online);// merge both arrays into one
            var keys = {};

            // build an index of keys
            for (var i = 0; i < offline.length; i += 1) {
                if (keys.hasOwnProperty(offline[i].id)) {
                    var orig = keys[offline[i].id];
                    // overwriting all properties or just pulling in offline status
                    if (favorOnline) {
                        orig.offline = offline[i].offline;
                        orig.offlineSince = offline[i].offlineSince;
                    } else {
                        angular.extend(orig, offline[i]);
                    }
                    offline.splice(i, 1);
                    i -= 1;
                } else {
                    keys[offline[i].id] = offline[i];
                }
            }

            return offline;
        }

        function selectAndFixResponseData(response) {
            return utils.fixObjGraph(response.data);
        }
    }
})();
```

Listing B.6: jobs/job.service.js

```
(function () {
    'use strict';


    angular.module('app').config(function ($stateProvider, $urlRouterProvider) {
        $urlRouterProvider.otherwise('/jobs');




        $stateProvider
            .state('jobs', {
```

```
        url: '/jobs',
        abstract: true,
        templateUrl: '/app/jobs/jobs.html',
        controllerAs: 'vm',
        controller: function ($state, $scope, toastr, jobService) {
            jobService.getAll(true).then(function (jobs) {
                $scope.jobs = jobs;
                return jobs;
            }, function (e) {
                toastr.error('An error occured while getting jobs.', 'Error',
                    e);
            });
        },
        resolve: {
            identity: function (principal) { return principal.identity(); }
        },
    })
    .state('jobs.list', {
        url: '',
        templateUrl: '/app/jobs/jobs.list.html'
    })

    .state('job', {
        url: '/jobs/{jobId:[0-9]+}',
        abstract: true,
        templateUrl: '/app/jobs/job.layout.html',
        controllerAs: 'vm',
        controller: function (job, identity, $scope, jobService) {
            var vm = this;
            $scope.job = job;

            updateLockedBy(vm, job, jobService, identity);
            $scope.$watch('job.lockedBy', function () {
                updateLockedBy(vm, job, jobService, identity);
            });

        },
        resolve: {
            identity: function (principal) { return principal.identity(); },
            job: function ($q, $stateParams, $state, jobService, utils) {
                var jobId = utils.coerceToInt($stateParams.jobId);
                if (!jobId) { return $q.reject('job id is required.'); }

                if ($state.job && $state.job.id === jobId) { return $q.when(
                    $state.job); }

                return jobService.getById(jobId).then(function (job) { return
                    ($state.job = job); });
            }
        },
    }).state('job.details', {
        url: '',
        templateUrl: '/app/jobs/job.details.html',
        controllerAs: 'vm',
        controller: function (job, identity, jobService) {
            var vm = angular.extend(this, job, {
                    error: null
                });
                updateLockedBy(vm, job, jobService, identity);
            }
        }).state('job.checkout', {
            url: '/checkout',
            templateUrl: '/app/jobs/job.checkout.html',
            controllerAs: 'vm',
            controller: 'CheckoutController',
            resolve: {
                action: function () { return 'checkout'; }
            }
        }).state('job.checkin', {
            url: '/checkin',
            templateUrl: '/app/jobs/job.checkin.html',
            controllerAs: 'vm',
            controller: 'CheckoutController',
            resolve: {
                action: function () { return 'checkin'; }
            }
        }).state('job.abandon', {
            url: '/abandon',
            templateUrl: '/app/jobs/job.abandon.html',
            controllerAs: 'vm',
            controller: 'CheckoutController',
            resolve: {
                action: function () { return 'abandon'; }
            }
        });

    function updateLockedBy(vm, job, jobService, identity) {
        vm.isLocked = jobService.isLocked(job);
        vm.lockedByOther = jobService.lockedByOther(job, identity);
        vm.lockedByMe = jobService.lockedByMe(job, identity);
    }

    });
})();
```

Listing B.7: jobs/jobs.js

B.3 Authentication

```
(function () {
    'use strict';

    var componentId = 'authenticationService';

    angular.module('app').factory(componentId, authenticationService);


    /**
     * @ngdoc object
     * @name app.authenticationService
```

```
 * @description
 *
 * Manages communication with the server regarding authentication
 * information. This service is also responsible to handling
 * offline user authentication.
 */
function authenticationService($q, $timeout, $http, $log) {
    var service = {
        identify: identify,
        login: login,
        logoff: logoff
    };

    return service;

    /**
     * @ngdoc method
     * @name app.authenticationService#identify
     * @methodOf app.authenticationService
     * @description
     *
     * Pings the server with a request for credential information
     * as the server knows. The end point of this request is
     * authenticated, so, if the browser submits an existing auth
     * cookie back with this request, the result will be the
     * authenticated username for the auth cookie used.  This
     * empowers session continuation between browser reloads.
     *
     * @param {string} username The username to login under.

     */
    function identify() {
        return $http
            .get('api/auth/identify')
            .then(function (response) {
                $log.debug(response);
                return { username: response.data, authenticated: true };
            }, function (response) {
                $log.debug(response);
                return $q.reject(response);
            });
    }

    function login(credentials) {
        return $http.post('api/auth/in', credentials)
        .then(function (r) {
            return (r.data === true) ?
                { username: credentials.username, authenticated: true } :
                $q.reject(credentials);
        }, function (e) {
            // operating offline??
            if (e.status === -1) {
                return { username: credentials.username, authenticated: true };
            }
            return $q.reject(e);
        });
        }

        function logoff() {
            return $http.post('api/auth/out');
        }
    }
})();
```

Listing B.8: auth/authentication.service.js

```
(function () {
    'use strict';

    var componentId = 'LoginController';

    angular.module('app').controller(componentId, LoginController);


    /**
     * @ngdoc controller
     * @name app.LoginController
     * @description
     *
     * Oversees the login process.
     *
     * In this demonstration, the authentication is limited to only
     * notifying the server what the active username is.  We are not
     * interested in a full blown user account or profile and passwords.
     */
    function LoginController($q, authenticationService, toastr, username, close) {
        var vm = angular.extend(this, {
            username: username,
            login: login
        });

        authenticationService
            .identify()
            .then(sayHi)
            .then(close);
        /**
         * @ngdoc method
         * @name app.LoginController#login
         * @methodOf app.LoginController
         * @description
         *
         * Attempts to login through the
         * {@link app.authenticationService authenticationService} and
         *  close on success.
         *
         * @param {string} username The username to login under.

         */
        function login(username) {
            return authenticationService
```

```
                .login({ username: username })
                .then(sayHi)
                .then(close);
        }

        function sayHi(u) {
            toastr.success('Welcome ' + u.username);
            return u;
        }
    }
})();
```

Listing B.9: auth/login.js

```
(function () {
    'use strict';

    var componentId = 'loginService';

    angular.module('app').factory(componentId, loginService);

    /**
     * @ngdoc object
     * @name app.loginService
     * @description
     *
     * Acquires credentials to submit for authentication.
     */
    function loginService($q, $uibModal) {
        var service = {
            login: prompt
        };

        return service;

        /**
         * @ngdoc object
         * @name app.loginService#login
         * @methodOf app.loginService
         * @description
         *
         * Acquires credentials to submit for authentication.
         *
         * This implementation uses Bootstrap to present a modal login
         * dialog prompt.
         *
         * @param {object} [seed] Provides seed values to the user
         * credential acquisition process.
         *
         * @returns {Q} A promise which resolves with authenticated
         * user credentials.
         */
        function prompt(seed) {
            var inst = $uibModal.open({
                animation: true,
                templateUrl: '/app/auth/login.html',
                controller: 'LoginController as vm',
                size: 'sm',
                backdrop: 'static',
                resolve: {
                    username: function () {
                        return ((seed || {}).username || 'guest');
                    },
                    close: function () {
                        return function () {
                            return inst.close.apply(inst, arguments);
                        };
                    }
                }
            });



            return inst.result;
        }
    }
})();
```

Listing B.10: auth/login.service.js

```
(function () {
    'use strict';

    var componentId = 'principal',
        _identity = void 0;

    angular.module('app').factory(componentId, principal);

    /**
     * @ngdoc object
     * @name app.principal
     * @description
     *
     * Manages the authenticated user credentials.
     */
    function principal($q, loginService) {
        var service = {
            identity: identity
        };

        return service;

        /**
         * @ngdoc function
         * @name app.principal#identity
         * @methodOf app.principal
         * @description
         *
         * Get the authenticated user credentials.
         *
         * @param {boolean=} [flush=undefined] Flags principal to drop
         * cached credentials and acquire new credentials from the
         * loginService.
         */
        function identity(flush) {
            if (flush === true) { _identity = void 0; }


            return _identity || (_identity = loginService
                    .login({ username: 'guest' })
                    .then(setIdentity));


        }

        function setIdentity(identity) {
            return (_identity = $q.when(identity));
        }
    }
})();
```

Listing B.11: auth/principal.js

B.4 Assets

```
(function () {
    'use strict';

    angular.module('app').config(function ($stateProvider, $urlRouterProvider) {
        $urlRouterProvider.otherwise('/jobs');

        $stateProvider.state('job.assets', {
            url: '/assets',
            abstract: true,
            views: {
                'sidebar@job': {
                    templateUrl: '/app/assets/assets.sidebar.html',
                    controllerAs: 'vma',
                    controller: 'AssetMruController'
                },
                '': { template: '<ui-view/>' }
            },

            resolve: {
                identity: function (principal) { return principal.identity(); },
                //mru: function ($q) { return $q.when([{ id: 123, prefix: 'AHU', rank:
                     9, area: 'basement' }]); }
            },
        }).state('job.assets.list', {
            url: '',
            views: {
                '': {
                    templateUrl: '/app/assets/assets.list.html',
                    controller: 'AssetsListController',
                    controllerAs: 'vm',
                },
            },
        })



        .state('job.asset', {
            url: '/assets/{assetId:[0-9]+}',
            abstract: true,
            views: {
                'sidebar@job': {
                    templateUrl: '/app/assets/assets.sidebar.html',
                    controllerAs: 'vma',
                    controller: 'AssetMruController'
                },
                '': { template: '<ui-view/>' }
            },

            resolve: {
                identity: function (principal) { return principal.identity(); },
                asset: function (job, $q, $state, $stateParams, jobService, utils,
                    assetMruService) {
                    var assetId = utils.coerceToInt($stateParams.assetId);
                    if (!assetId) { return $q.reject('Asset id is required.'); }

                    if ($state.asset && $state.asset.id === assetId) {
                        assetMruService.push($state.asset, job);
                        return $q.when($state.asset);
                    }

                    return jobService.getAssetByJobAndId(job, assetId).then(function (
                        asset) {
                        if (!asset) { return $q.reject('asset not found'); }
                        assetMruService.push(asset, job);
                        return ($state.asset = asset);
                    });
                }
            },
        })
        .state('job.asset.details', {
            url: '',
            templateUrl: '/app/assets/asset.details.html',
            controllerAs: 'vm',
            controller: 'AssetDetailsController'
        });
    });

})();
```

Listing B.12: asets/assets/js

```
(function () {
```

```
'use strict';


var componentId = 'AssetDetailsController';


angular.module('app').controller(componentId, AssetDetailsController);


function AssetDetailsController(job, asset, $log, $state, $scope, toastr,
    jobService, offlineStore, utils) {
    var vm = angular.extend(this, {
        submit: submit,
        reset: resetForm,
        reconcile: false
    });

    if (job.reconcile === true) {
        vm.reconcile = true;
        offlineStore
            .get('asset', asset.id, true)
            .then(function (_) { $scope.current = _; });
    }
    $scope.asset = angular.copy(asset);



    function goHome(msg, whereFrom) {
        toastr.error(msg, whereFrom || 'Asset Details');
        $state.go('job.assets');
    }

    function submit(userForm, editedAsset) {
        delete editedAsset.job;
        if (vm.reconcile) {
            if($scope.current){
                editedAsset.rowVersion = $scope.current.rowVersion;
            }

        }
        jobService
            .save(job, editedAsset)
            .then(function (result) {
                angular.extend(asset, result);
                $scope.asset = angular.copy(asset);
                userForm.$setPristine();
                toastr.success('Changes saved.', 'Asset Details');
            }, function (e) {
                //toastr.error('Error while saving', 'Asset Details');
            });
    }

    function resetForm(userForm, editedAsset) {
        angular.extend(editedAsset, asset);
        userForm.$rollbackViewValue();
        userForm.$setPristine();
        }
    }


})();
```

Listing B.13: assets/asset.details.js

```
(function () {
    'use strict';

    var componentId = 'assetMruService', s_mru = {};

    angular
        .module('app').factory(componentId, assetMruService)
        .controller('AssetMruController', function (job, assetMruService) {
            var vm = angular.extend(this, {
                mru: assetMruService.mru(job)
            });
        });

    function assetMruService(utils) {
        var service = {
            mru: function (job) { return (s_mru[job.id] || (s_mru[job.id] = [])); },
            push: function (asset, job) { return push(asset, job, service.mru(job)); }
        };

        return service;

        function push(asset, job, mru) {
            var known = utils.findById(mru, asset.id);
            if (known) {
                mru.splice(mru.indexOf(known), 1);
            }
            mru.unshift(asset);

            return asset;
        }
    }
})();
```

Listing B.14: assets/asset.mru.js

```
(function () {
    'use strict';

    var componentId = 'AssetsListController', s_mru = {};

    angular.module('app').controller('AssetsListController', AssetsListController);

    function AssetsListController(job, $state, $scope, jobService, toastr) {
        var vm = angular.extend(this, {
            assetAreas: [],
            jobId: job.id,
            haveSearched: false,
            noAssetsAssigned: false,

            chosenArea: null
        });

        $scope.$watch('vm.chosenArea', findAssets);

        jobService
            .getAssetAreas(job)
            .then(function (_) {
                vm.assetAreas = _;
                if (!(_ && _.length)) {
                    // event empty string asset area will be returned, if there
                    // are no areas found, there are no assets to search on.
                    vm.noAssetsAssigned = true;
                }
            }, function (e) {
                returnToJob('Error while attempting to load asset areas.');
            });

        function findAssets() {
            if (!!vm.chosenArea) {
                jobService.getAssets(job, vm.chosenArea)
                    .then(function (assets) {
                        vm.haveSearched = true;
                        vm.assets = assets;
                        if (!assets) {
                            vm.noAssetsAssigned = true;
                        }
                    }, function (e) {
                        returnToJob('Unable to get assets');
                    });
            }
        }

        function returnToJob(reason) {
            toastr.error(reason, 'Assets');
            $state.go('job.details');
        }
    }
})();
```

Listing B.15: assets/assets.list.js

```
(function () {
    'use strict';

    angular.module('app').config(function ($stateProvider, $urlRouterProvider) {


        $stateProvider
            .state('assets', {
                abstract: true,
                url: '/{jobId:[0-9]+}/assets',
                templateUrl: '/app/assets/assets.html',
                controllerAs: 'vm',
                controller: function (areas, mru, assetsService, jobsService, $log,
                    $scope, $stateParams) {
                    var vm = angular.extend(this, {
                        job: null,
                        assets: [],
                        areas: areas,
                        mru: mru,
                        offline: null,
                        name:assetsService.name
                    });


                    jobsService.byId($stateParams.jobId).then(function (job) {
                        vm.job = job;
                        vm.offline = job.offline;
                    });

                    $scope.$watch('vm.assetType', findAssets);
                    $scope.$watch('vm.assetArea', findAssets);

                    function findAssets() {
                        if (!!vm.assetType || !!vm.assetArea) {
                            assetsService.byJobTypeArea(vm.job, vm.assetType, vm.
                                assetArea).then(function (assets) {
                                vm.assets = assets;
                            }, function (e) {
                                $log.error('Unable to get assets', e);
                            });
                        }
                    }
                },
                resolve: {
                    identity: function (principal) {
                        return principal.identity();
                    },
                    areas: function (assetsService, $stateParams) {
                        return assetsService.allAreas($stateParams.jobId);
                    },
                    mru: function () { return [];}
                    //assets: function ($stateParams, assetsService) {
                    //    return assetsService.byJobIdAndParent($stateParams.jobId);
                    //},
                }
            })
        .state('assets.list', {
            url: '',
            views: {
                'filter': {
                    templateUrl: '/app/assets/assets.filter.html',
                },
                'list': {
                    templateUrl: '/app/assets/assets.list.html',
                }
            }
        })
        .state('assets.detail', {
            url: '/{assetId:[0-9]+}',
            templateUrl: '/app/assets/assets.detail.html',
            controller: function (mru, utils, assetsService, $stateParams) {
                var vm = angular.extend(this, {
                    name: assetsService.name,
                    asset:null
                });

                activate($stateParams.assetId, $stateParams.jobId);

                // check for recently used local version
                function activate(assetId, jobId) {
                    var asset = utils.findById(mru, assetId);
                    if (asset) {
                        var i = mru.indexOf(asset);
                        mru.splice(i, 1);
                        mru.unshift(asset);

                        return angular.extend(vm, asset, { asset: asset });
                    }

                    return assetsService
                        .byJobAndId(jobId, assetId)
                        .then(function (asset) {
                            mru.unshift(asset);
                            angular.extend(vm, asset, { asset: asset });
                        });
                }
            }
        });
    });

})();
```

Listing B.16: assets/assets.route.config.js

```
(function () {
    'use strict';

    var componentId = 'AssetSidebarController';

    angular.module('app').controller(componentId, AssetSidebarController);



    function AssetSidebarController($stateParams, $q, jobService, $log) {
        var vm = angular.extend(this, {
            jobName: '',
            jobId: NaN
        });
        vm.jobId = resolveJobId($stateParams, $q);
        jobService.getById(vm.jobId)
        .then(function (job) {
            vm.jobName = job.name;
        }, function (e) {
            $log.warning('Error while getting job for asset sidebar nav', e);
        });
    }


    // resolves a job id from route state parameters
    function resolveJobId($stateParams, $q) {
        var jobId = $stateParams.jobId;
        if (angular.isNumber(jobId)) {
            return jobId;
        } else if (!jobId) {
            $q.reject('jobId is required');
        } else if (angular.isString(jobId)) {
            return parseInt(jobId);
        }
        return $q.reject('Invalid job id.' + jobId);
    }
})();
```

Listing B.17: assets/assets.sidebar.js

APPENDIX C

SERVER CODE

The following represents of core functional code to the approach above.

C.1 Change Set Code

```
namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Denotes the intended behavior when processing a <see cref="T:Rel.Data.Bulk.
        ChangeItem"/>.
    /// </summary>
    public enum ChangeAction
    {
        /// <summary>
        ///   Identifies that a change is meant to initialize a
        ///   domain. Only used when sending changes to client to
        ///   establish baseline entity states.
        /// </summary>
        Initialize,

        /// <summary>
        ///   Specifies the change is meant to create a new entity.
        ///   Requires AFIM
        /// </summary>
        Create,

        /// <summary>
        ///   Specifies the change is meant to update an entity.
        ///   Requires BFIM and AFIM.
        /// </summary>
        Update,

        /// <summary>
        ///   Specifies the change is meant to delete an entity.
        ///   Requires BFIM.
        /// </summary>
        Delete,
    }
}
```

Listing C.1: src/Rel.Data/Bulk/ChangeAction.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Serves as a common ancestor for generic implementation such
    ///   that a single collection may contain multiple change item
    ///   generic types.
    /// </summary>
    [CustomValidation(typeof(ChangeValidator), "SanityCheck")]
    public abstract class ChangeItem
    {
        private ICollection<ValidationResult> _validationResults =
            new List<ValidationResult>();

        /// <summary>
        ///   Gets or sets the action intended to result from this change.
        /// </summary>
        /// <value>The action.</value>
        public ChangeAction Action { get; set; }

        /// <summary>
        ///   Gets the validation results of this change.
        /// </summary>
        /// <value>The validation results.</value>
        protected internal ICollection<ValidationResult> ValidationResults
        { get { return _validationResults; } }

        /// <summary>
        ///   Gets the type of the entity.
        /// </summary>
        /// <returns></returns>
        public abstract Type GetEntityType();

        /// <summary>
        ///   Gets the after completion image of this change.
        /// </summary>
        /// <returns></returns>
        internal abstract object GetAFIM();

        /// <summary>
        ///   Gets the before completion image of this change..
        /// </summary>
        /// <returns></returns>
        internal abstract object GetBFIM();
    }

    /// <summary>
    ///   Provides concrete instancing for generic change items. This
    ///   generic version makes client communication simple by
    ///   strongly enforcing an endpoint for data of type TEntity in a <see cref="
        ChangeSet"/>.
    /// </summary>
    /// <typeparam name="TEntity">The type of the entity.</typeparam>
```

```
public class ChangeItem<TEntity> : ChangeItem
{
    internal static readonly IEnumerable<ChangeItem<TEntity>> Empty = new
        ChangeItem<TEntity>[0];

    /// <summary>
    ///   Initializes a new instance of the
    ///   <see cref="ChangeItem{TEntity}"/> class.
    /// </summary>
    public ChangeItem()
    {
    }

    /// <summary>
    ///   Initializes a new instance of the
    ///   <see cref="ChangeItem{TEntity}"/> class.
    /// </summary>
    /// <param name="bfim">The bfim.</param>
    /// <param name="afim">The afim.</param>
    public ChangeItem(TEntity bfim, TEntity afim)
    {
        BFIM = bfim;
        AFIM = afim;
    }

    /// <summary>
    ///   Initializes a new instance of the
    ///   <see cref="ChangeItem{TEntity}"/> class.
    /// </summary>
    /// <param name="action">The action.</param>
    /// <param name="bfim">The bfim.</param>
    /// <param name="afim">The afim.</param>
    public ChangeItem(ChangeAction action, TEntity bfim, TEntity afim)
        : this(bfim, afim)
    {
        Action = action;
    }

    /// <summary>
    ///   Gets or sets the after image of this change.
    /// </summary>
    /// <value>The after image.</value>
    public TEntity AFIM { get; set; }

    /// <summary>
    ///   Gets or sets the before image of this change.
    /// </summary>
    /// <value>The before image.</value>
    public TEntity BFIM { get; set; }

    /// <summary>
    ///   Gets the type of the entity.
    /// </summary>
    /// <returns>
    ///   The type of entity associated with this change.
    /// </returns>
```

```
        public override Type GetEntityType()
        {
            return typeof(TEntity);
        }

        /// <summary>
        ///   Gets the after completion image of this change.
        /// </summary>
        /// <returns></returns>
        internal override object GetAFIM()
        {
            return AFIM;
        }

        /// <summary>
        ///   Gets the before completion image of this change..
        /// </summary>
        /// <returns></returns>
        internal override object GetBFIM()
        {
            return BFIM;
        }
    }
}
```

Listing C.2: src/Rel.Data/Bulk/ChangeItem.cs

```
using Rel.Data.Models;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Curries a series of actions to be performed as an atomic action.
    /// </summary>
    public class ChangeSet //: IEnumerable<ChangeItem>
    {
        /// <summary>
        ///   ChangeSet's flavor of <see langword="null"/>.
        /// </summary>
        internal static readonly ChangeSet Empty = new ChangeSet();

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="ChangeSet"/> class.
        /// </summary>
        public ChangeSet()
        {
            Assets = new List<ChangeItem<Asset>>();
        }

        /// <summary>
        ///   Gets or sets the assets.
        /// </summary>
        /// <value>The assets.</value>
        public List<ChangeItem<Asset>> Assets { get; set; }

        /// <summary>
        ///   Gets a value indicating whether this instance is empty.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if this instance is empty;
        ///   otherwise, <see langword="false"/>.
        /// </value>
        public bool IsEmpty { get { return Assets == null || Assets.Count == 0; } }

        /// <summary>
        ///   Gets the total number of items in this change set.
        /// </summary>
        /// <value>The total items count.</value>
        /// <remarks>
        ///   The current implementation is limited to only one real
        ///   underlying type of
        ///   <see cref="T:Rel.Data.Models.Asset"/>. This property
        ///   will prove more useful as the project grows over time
        ///   and additional entity types are added with support for
        ///   bulk update.
        /// </remarks>
        [Range(1, int.MaxValue)]
        public long TotalItemsCount { get { return Assets.Count; } }
    }
}
```

Listing C.3: src/Rel.Data/Bulk/ChangeSet.cs

```
using Rel.Data.Configuration;
using Rel.Data.Diagnostics;
using Rel.Data.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Transactions;

namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Manages integration of a collection of changes
    ///   into an <see cref="Rel.Data.IDataContext"/>.
    /// </summary>
    public class ChangeSetProcessor
    {
        private readonly IDataContext _db;
        private readonly bool _disableResolution = false;
        private readonly IConflictResolver _resolver;
        private readonly string _resolverName;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="ChangeSetProcessor"/> class.
        /// </summary>
        /// <param name="context">The data context.</param>
        /// <param name="conflictResolver">The conflict resolver.</param>
        public ChangeSetProcessor(IDataContext context, IConflictResolver
            conflictResolver)
        {
            _db = context;
            _resolver = conflictResolver;
            _resolverName = conflictResolver.GetType().Name;
            if (conflictResolver is RejectConcurrentEditsConflictResolver)
                _disableResolution = true;
        }

        /// <summary>
        ///   Builds the initial change set which a client will build
        ///   changes from.
        /// </summary>
        /// <param name="partitionId">The partition identifier.</param>
        /// <returns>
        ///   A change set where every item is marked as Initialize.
        /// </returns>
        public ChangeSet BuildInitialChangeSet(int partitionId)
        {
            var cs = new ChangeSet();
            cs.Assets = _db
                .Assets
                .GetAll()
                .Where(_ => _.JobId == partitionId)
                .ToList()
                .Select(_ => new ChangeItem<Asset>(null, _) { Action = ChangeAction.
                    Initialize })
                .ToList();

            return cs;
        }

        /// <summary>
        ///   Processes the specified change set under the given
        ///   partition id.
        /// </summary>
        /// <param name="partitionId">The partition identifier.</param>
        /// <param name="claimIt">
        ///   Set to <see langword="true"/> to indicate desire to
        ///   claim exclusive access to the partition if unable to
        ///   commit in a single optimistic concurrency control update.
        /// </param>
        /// <param name="changeSet">The change set.</param>
        /// <returns>
        ///   ChangeSet.Empty if the change set was accepted;
        ///   otherwise, a new change set is returned containing a
        ///   "migration script" for the submitter to bring their
        ///   local data current with the remote version for the items submitted.
        /// </returns>
        /// <exception cref="System.ArgumentNullException">
        ///   changeSet
        /// </exception>
        /// <exception cref="EntityNotFoundException"></exception>
        /// <exception cref="ConcurrencyException"></exception>
        public ChangeSet Process(int partitionId, bool claimIt, ChangeSet changeSet)
        {
            if (changeSet == null)
                throw new ArgumentNullException("changeSet");

            var result = ChangeSet.Empty;
            using (var perfScope = new ChangeSetPerformanceScope(_resolverName,
                changeSet))
            {
                using (perfScope.TimeReplay())
                    Replay(changeSet);

                if (!_db.Validate())
                    throw Error.InvalidData(changeSet, false);

                bool accepted;

                using (perfScope.TimeSave())
                    accepted = Apply(partitionId, claimIt, changeSet);

                if (accepted)
                {
                    perfScope.Complete();
                }
                else
                {
                    using (perfScope.TimeRedress())
                    {
                        bool resolved = false;
                        using (perfScope.TimeCacheBuilding())
                            FlushAndReCache();
                        using (perfScope.TimeResolve())
                            resolved = ResolveConflicts(changeSet);

                        if (resolved)
                        {
                            if (!_db.Validate())
                                throw Error.InvalidData(changeSet, true);

                            using (perfScope.TimeSave())
                                accepted = Apply(partitionId, claimIt, changeSet);

                            if (accepted)
                            {
                                perfScope.Complete();
                                return result;
                            }
                        }

                        result = BuildReconciliationChangeSet(changeSet);
                    }
                }
            }
            return result;
        }

        private bool Apply(int partitionId, bool claimIt, ChangeSet changeSet)
        {
            ILock @lock;
            bool accepted = false;

            using (var scope = new TransactionScope(TransactionScopeOption.RequiresNew
                ))
            {
                @lock = _db.Jobs.GetAll().Where(_ => _.Id == partitionId).
                    SingleOrDefault();

                if (@lock == null)
                    throw new EntityNotFoundException();

                if (@lock.Status == LockStatus.Closed)
                {
                    _db.RejectChanges();
                    throw Error.PessimisticLock();
                }

                try
                {
                    _db.AcceptChanges();
                    scope.Complete();
                    accepted = true;
                }
                catch (ConcurrencyException)
                {
                    // normal OC update failed, claim partition if
                    // necessary then fall back to reconciliation.
                }
            }

            if (accepted)
            {
                if (@lock.Status == LockStatus.Exclusive)
                {
                    @lock.Open();
                    _db.AcceptChanges();
                }
            }
            else
            {
                _db.RejectChanges();

                if (@lock.Status == LockStatus.Open &&
                    (claimIt ||
                    IsOfSufficientSize(changeSet)))
                {
                    using (var scope = new TransactionScope(TransactionScopeOption.
                        RequiresNew))
                    {
                        try
                        {
                            @lock.Close();
                            _db.AcceptChanges();
                            scope.Complete();
                        }
                        catch (ConcurrencyException)
                        {
                            throw Error.PessimisticLock();
                        }
                    }
                }
            }
            return accepted;
        }

        /// <summary>
        ///   Builds the reconciliation change set.
        /// </summary>
        /// <param name="changeSet">
        ///   The change set containing unresolvable conflict(s).
        /// </param>
        /// <returns>
        ///   A reconciliation change set fit for client consumption.
        /// </returns>
        private ChangeSet BuildReconciliationChangeSet(ChangeSet changeSet)
        {
            var reconcile = new ChangeSet();

            reconcile.Assets = CreateReconcileChangeCollection(_db.Assets, changeSet.
                Assets);

            return reconcile;
        }

        /// <summary>
        ///   Creates a primary key join filter expression.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <typeparam name="TKey">The type of the key.</typeparam>
        /// <param name="keySelector">The key selector.</param>
        /// <param name="keys">The keys.</param>
        /// <returns>
        ///   A LINQ expression which will filter on a known
        ///   collection of primary key values.
        /// </returns>
        /// <remarks>
        ///   Note that this builds and returns a LINQ expression, not
        ///   a function. This is necessary for the LINQ to SQL
        ///   provider (or any other provider which may intelligently
        ///   bridge languages) to translate a LINQ query to SQL query.
        /// </remarks>
        private Expression<Func<TEntity, bool>> CreateJoinFilter<TEntity, TKey>(
            Expression<Func<TEntity, TKey>> keySelector, IEnumerable<TKey> keys)
        {
            var lparams = Expression.Parameter(typeof(TEntity), "e");
            return Expression.Lambda<Func<TEntity, bool>>(
                Expression.Call(
                    typeof(Enumerable),
                    "Contains",
                    new[] { typeof(TKey) },
                    new Expression[] { Expression.Constant(keys, typeof(IEnumerable<
                        TKey>)), Expression.Invoke(keySelector, lparams) }
                ),
                lparams
                );
        }

        /// <summary>
        ///   Creates a collection of changes to be applied remotely
        ///   for reconciliation.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <typeparam name="TKey">The type of the key.</typeparam>
        /// <param name="repository">The repository.</param>
        /// <param name="changes">The changes.</param>
        /// <returns>
        ///   A collection of changes necessary to bring the submitter
        ///   of the changes which failed to current for all items submitted.
        /// </returns>
        private List<ChangeItem<TEntity>> CreateReconcileChangeCollection<TEntity,
            TKey>(IRepository<TEntity, TKey> repository, IEnumerable<ChangeItem<
            TEntity>> changes)
        {
            List<ChangeItem<TEntity>> reconcile = new List<ChangeItem<TEntity>>();
            foreach (var change in changes)
            {
                if (change.Action == ChangeAction.Create)
                    // for the time being, create is not capable of
                    // conflict or reconciliation
                    continue;

                var updateTo = repository.GetById(repository.GetId(change.BFIM));

                if (updateTo == null)
                {
                    reconcile.Add(new ChangeItem<TEntity>(change.BFIM, default(TEntity
                        ))
                    {
                        Action = ChangeAction.Delete
                    });
                }
                else
                {
                    reconcile.Add(new ChangeItem<TEntity>(change.BFIM, updateTo)
                    {
                        Action = ChangeAction.Update
                    });
                }
            }

            return reconcile;
```

```
}

private void FlushAndReCache()
{
    // get rid of all change set changes in the local cache
    // and pull down fresh copies of cache items.
    using (var scope = new TransactionScope(TransactionScopeOption.RequiresNew
        ))
    {
        FlushAndReCache(_db.Assets);
        scope.Complete();
    }
}

/// <summary>
///   Flushes the local cache of the given repository and
///   re-queries the GetAll() method to load up fresh copies
///   of data.
/// </summary>
/// <typeparam name="TEntity">The type of the entity.</typeparam>
/// <typeparam name="TKey">The type of the key.</typeparam>
/// <param name="repository">The repository.</param>
/// <remarks>
///   This method is necessary as the initial attempt to
///   submit a change set is not proactive in fetching items
///   from the underlying data store. This is predicated on
///   the assumption that majority of change set submissions
///   should succeed on their initial pass and be "clean".
///   Since the items are not preloaded, the local data cache
///   does not actually know the current state of any items it holds.
///   <para>
///     This method will record the identities of all items it
///     holds locally, empty the local cache, then query the
///     underlying data store for all identities previously witnessed.
///   </para>
/// </remarks>
private void FlushAndReCache<TEntity, TKey>(IRepository<TEntity, TKey>
    repository)
{
    var ids = repository.Select(repository.GetId).ToArray();
    repository.Flush();
    var e = CreateJoinFilter(repository.KeySelector, ids);
    repository.GetAll().Where(e.Compile()).ToList();
}

/// <summary>
///   Determines whether the change set is of sufficient size
///   to warrant attempting exclusive access.
/// </summary>
/// <param name="changeSet">The change set.</param>
/// <returns></returns>
private bool IsOfSufficientSize(ChangeSet changeSet)
{
    var threshold = DataConfigurationSection.Default.ChangeSets.LockThreshold;
    if (threshold > 0 && changeSet.TotalItemsCount >= threshold)
        return true;
    return false;
}

/// <summary>
///   Replays actions recorded in a change set on a repository.
/// </summary>
/// <typeparam name="TEntity">The type of the entity.</typeparam>
/// <typeparam name="TKey">The type of the key.</typeparam>
/// <param name="repository">The repository.</param>
/// <param name="changes">The changes.</param>
/// <exception cref="System.InvalidOperationException">
///   Initialize action only supported on client. or Invalid
///   change action.
/// </exception>
/// <exception cref="System.NotImplementedException"></exception>
private void ProcessActions<TEntity, TKey>(IRepository<TEntity, TKey>
    repository, IEnumerable<ChangeItem<TEntity>> changes)
{
    foreach (var item in changes)
    {
        switch (item.Action)
        {
            case ChangeAction.Initialize:
                throw new InvalidOperationException("Initialize action only
                    supported on client.");
            case ChangeAction.Create:
                repository.Create(item.AFIM);
                break;

            case ChangeAction.Update:
                repository.Update(item.AFIM);
                break;

            case ChangeAction.Delete:
                repository.Delete(item.BFIM);
                break;

            default:
                throw new InvalidOperationException("Invalid change action.");
        }
    }
}

private void Replay(ChangeSet changeSet)
{
    ProcessActions(_db.Assets, changeSet.Assets);
}

/// <summary>
///   Attempts to resolve all conflicts in the change set given.
/// </summary>
/// <param name="changeSet">The change set.</param>
/// <returns>
///   <see langword="true"/> if all conflicts were resolved;
///   otherwise, <see langword="false"/>.
```

```
        /// </returns>
        private bool ResolveConflicts(ChangeSet changeSet)
        {
            if (_disableResolution) return false;
            var lookup = _db.Assets.ToDictionary(_ => _.Id);

            // attempt to resolve all conflicts
            bool isClean = changeSet.Assets.All(_ =>
                _resolver.Resolve(_db.Assets, _, lookup));
            return isClean;
        }
    }
}
```

Listing C.4: src/Rel.Data/Bulk/ChangeSetProcessor.cs

```
using System.ComponentModel.DataAnnotations;

namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Inspects that a change is well formed. The change may still
    ///   result in invalid data when applied.
    /// </summary>
    /// <typeparam name="TEntity">The type of the entity.</typeparam>
    public static class ChangeValidator
    {
        /// <summary>
        ///   Provides a basic sanity check that a change item is well formed.
        /// </summary>
        /// <param name="item">The item.</param>
        /// <param name="context">The context.</param>
        /// <returns>
        ///   <see cref="P:System.ComponentModel.DataAnnotations.ValidationResult.
            Success"/>
        ///   if the change is valid; otherwise, a ValidationResult
        ///   denoting the reason the change failed validation.
        /// </returns>
        public static ValidationResult SanityCheck(ChangeItem item, ValidationContext
            context)
        {
            switch (item.Action)
            {
                case ChangeAction.Create:
                    if (item.GetBFIM() != null)
                        return new ValidationResult("BFIM not permitted for new
                            entries.", new[] { "BFIM" });
                    if (item.GetAFIM() == null)
                        return new ValidationResult("AFIM required for new entries.",
                            new[] { "AFIM" });
                    break;

                case ChangeAction.Update:
                    if (item.GetBFIM() == null)
```

```
                        return new ValidationResult("BFIM required for update entries.
                            ", new[] { "BFIM" });
                    if (item.GetAFIM() == null)
                        return new ValidationResult("AFIM required for update entries.
                            ", new[] { "AFIM" });
                    break;

                case ChangeAction.Delete:
                    if (item.GetBFIM() == null)
                        return new ValidationResult("BFIM required for new entries.",
                            new[] { "BFIM" });
                    if (item.GetAFIM() != null)
                        return new ValidationResult("AFIM not permitted for new
                            entries.", new[] { "AFIM" });
                    break;

                default:
                    return new ValidationResult("Invalid change action", new[] { "
                        Action" });
            }

            return ValidationResult.Success;
        }
    }
}
```

Listing C.5: src/Rel.Data/Bulk/ChangeValidator.cs

```
using System.Collections.Generic;

namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Identifies a concurrent, conflicting update resolution
    ///   strategy provider.
    /// </summary>
    public interface IConflictResolver
    {
        /// <summary>
        ///   Attempts to resolve a conflict by inspecting the past,
        ///   current, and divergent state of an entity.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <typeparam name="TKey">The type of the key.</typeparam>
        /// <param name="repository">The repository.</param>
        /// <param name="change">The change.</param>
        /// <param name="index">
        ///   The index into repository for O(1) lookup.
        /// </param>
        /// <returns></returns>
        bool Resolve<TEntity, TKey>(IRepository<TEntity, TKey> repository, ChangeItem<
            TEntity> change, IDictionary<TKey, TEntity> index);
    }
}
```

Listing C.6: src/Rel.Data/Bulk/IConflictResolver.cs

```
using Rel.Merge;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Resolves conflicting writes using merge strategies.
    /// </summary>
    /// <remarks>
    ///   Logic considerations
    ///
    ///     |       HAS VALUE       |
    ///   # | BFIM | Current | AFIM | Represents
    ///   --+------+---------+-------------------------------------------
    ///   0 |  T   |    T    |  T   | normal OCC update
    ///   1 |  T   |    T    |  F   | normal OCC delete
    ///   2 |  T   |    F    |  T   | dirty deleted OCC update
    ///   3 |  T   |    F    |  F   | dirty deleted OCC delete
    ///   4 |  F   |    T    |  T   | dirty created OCC create [1]
    ///   5 |  F   |    T    |  F   | n/a [2]
    ///   6 |  F   |    F    |  T   | Normal create
    ///   7 |  F   |    F    |  F   | n/a [2]
    ///
    ///   [1] Not considered as possible in this version as no active
    ///   logic is given for how to find a possible collision between
    ///   entities beyond ID value. The ID value will never collide
    ///   because all IRepository{,}.Create calls replace any current
    ///   value for the ID with a new one upon save.
    ///
    ///   [2] Not considered as possible in this version as if there
    ///   is BFIM and no AFIM in a change, then there is nothing to
    ///   map the change back to a specific entity.
    /// </remarks>
    public class MergeConcurrentEditsConflictResolver : IConflictResolver
    {
        private readonly IMergeProvider _merge;

        /// <summary>
        ///   Initializes a new instance of the <see
        ///   cref="MergeConcurrentEditsConflictResolver"/> class.
        /// </summary>
        /// <param name="merge">The merge provider.</param>
        public MergeConcurrentEditsConflictResolver(IMergeProvider merge)
        {
            _merge = merge;
        }

        /// <summary>
        ///   Attempts to resolve a conflict by inspecting the past,
        ///   current, and divergent state of an entity.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <typeparam name="TKey">The type of the key.</typeparam>
        /// <param name="repository">The repository.</param>
        /// <param name="change">The change.</param>
        /// <returns>
        ///   <see langword="true"/> if successfully resolved conflict
        ///   with merge; otherwise, <see langword="false"/>.
        /// </returns>
        public bool Resolve<TEntity, TKey>(IRepository<TEntity, TKey> repository,
            ChangeItem<TEntity> change, IDictionary<TKey,TEntity> index)
        {
            var action = change.Action;
            TKey id;
            TEntity current;
            switch (action)
            {
                case ChangeAction.Create:
                    id = repository.GetId(change.AFIM);
                    repository.Create(change.AFIM);
                    return true;

                case ChangeAction.Update:
                case ChangeAction.Delete:

                    var kind = MergeKind.Auto;
                    id = repository.GetId(change.BFIM);
                    if (!index.TryGetValue(id, out current))
                    {
                        //kind = MergeKind.HiddenDelete;
                        //current = repository.GetById(id);
                    }
                    var resolution = _merge.Merge(kind, change.BFIM, current, change.
                        AFIM);

                    if (resolution.IsResolved())
                    { // merge approved of delete
                        ReflectMergeInRepo(repository, change, current, resolution);
                        return true;
                    }
                    break;

                default:
                    break;
            }

            return false;
        }

        /// <summary>
        ///   Reflects the merge in repo.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <typeparam name="TKey">The type of the key.</typeparam>
        /// <param name="repository">The repository.</param>
```

```
        /// <param name="change">The change.</param>
        /// <param name="resolution">The resolution.</param>
        /// <exception cref="System.InvalidOperationException"></exception>
        private void ReflectMergeInRepo<TEntity, TKey>(IRepository<TEntity, TKey>
            repository, ChangeItem<TEntity> change, TEntity current, IMergeResolution<
            TEntity> resolution)
        {
            switch (resolution.Result)
            {
                case MergeActionResult.Resolved:
                    // absence of modifiers denotes the merge is wholly
                    // resolved by the resolution framework and does not
                    // need any action taken from the operational context.
                    break;

                case MergeActionResult.Delete:
                    repository.Delete(current);
                    break;

                case MergeActionResult.Create:
                    repository.Create(resolution.ResolvedValue);
                    break;

                case MergeActionResult.Update:
                    repository.Update(resolution.ResolvedValue);
                    break;

                case MergeActionResult.Unresolved:
                default:
                    throw new InvalidOperationException();
            }
        }
    }
}
```

Listing C.7: src/Rel.Data/Bulk/MergeConcurrentEditsConflictResolver.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Rel.Data.Bulk
{
    /// <summary>
    ///   Precludes concurrent edits of data. All updates must be
    ///   performed on current values.
    /// </summary>
    public sealed class RejectConcurrentEditsConflictResolver: IConflictResolver
    {
        /// <summary>
        ///   Rejects all concurrent updates.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <typeparam name="TKey">The type of the key.</typeparam>
        /// <param name="repository">The repository.</param>
        /// <param name="change">The change.</param>
        /// <returns><see langword="false"/> always.</returns>
        public bool Resolve<TEntity, TKey>(IRepository<TEntity, TKey> repository,
            ChangeItem<TEntity> change, IDictionary<TKey, TEntity> index)
        {
            return false;
        }
    }
}
```

Listing C.8: src/Rel.Data/Bulk/RejectConcurrentEditsConflictResolver.cs

C.2 Business DAL Scafold

```
using Rel.Data.Models;
using System;

namespace Rel.Data
{
    /// <summary>
    ///   Represents a composite data store for Thesis Portal entities.
    /// </summary>
    public interface IDataContext
        : IDisposable
    {
        /// <summary>
        ///   Gets the asset repository.
        /// </summary>
        /// <value>The assets.</value>
        IRepository<Asset, int> Assets { get; }

        /// <summary>
        ///   Gets the job repository.
        /// </summary>
        /// <value>The jobs.</value>
        IRepository<Job, int> Jobs { get; }

        /// <summary>
        ///   <para>
        ///     Attempts to push any changes made in this data context
        ///     since the last time it communicated with the
        ///     underlying data store to persisted storage.
        ///   </para>
        ///   <para>
        ///     Any error preventing a complete commit of the changes
        ///     within throws an exception.
        ///   </para>
        /// </summary>
        void AcceptChanges();

        /// <summary>
        ///   Undoes any changes made in this data context since the
        ///   last time it communicated with the underlying data store.
        /// </summary>
        void RejectChanges();

        /// <summary>
        ///   Validates changes in this data context.
        /// </summary>
        /// <returns>
        ///   <see langword="true"/> if valid; otherwise, false.
        /// </returns>
        bool Validate();
    }
}
```

Listing C.9: src/Rel.Data/IDataContext.cs

```
namespace Rel.Data
{
    /// <summary>
    ///   Represents a pessimistically lockable data block.
    /// </summary>
    public interface ILock
    {
        /// <summary>
        ///   Gets the current status of this lock.
        /// </summary>
        /// <value>The status.</value>
        LockStatus Status { get; }

        /// <summary>
        ///   Attempts migrate access from Open to Exclusive.
        /// </summary>
        /// <returns>The final state of the lock.</returns>
        LockStatus Close();

        /// <summary>
        ///   Attempts to release this lock from Exclusive to Open status.
        /// </summary>
        /// <returns>The final state of the lock.</returns>
        LockStatus Open();
    }
}
```

Listing C.10: src/Rel.Data/ILock.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;

namespace Rel.Data
{
    /// <summary>
    ///   Represents a local cache and interface to an underlying data
    ///   store of TEntity types.
    /// </summary>
    /// <typeparam name="TEntity">The type of the entity.</typeparam>
    /// <typeparam name="TKey">The type of the key.</typeparam>
    public interface IRepository<TEntity, TKey> : IEnumerable<TEntity>, IQueryable<
        TEntity>
    {
        /// <summary>
        ///   Gets an expression which, when compiled and run, yields
        ///   the primary key to entities of type TEntity.
        /// </summary>
        /// <value>The key selector.</value>
        Expression<Func<TEntity, TKey>> KeySelector { get; }

        /// <summary>
        ///   Adds the specified entity to this repository and places
        ///   it in queue for creation when the parent data context
        ///   has changes on it accepted.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>
        ///   The entity given or a potential proxy instance for
        ///   interfacing with underlying framework.
        /// </returns>
        TEntity Create(TEntity entity);

        /// <summary>
        ///   Adds the specified entity to this repository and places
        ///   it in queue for deletion when the parent data context
        ///   has changes on it accepted.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>
        ///   The entity given or a potential proxy instance for
        ///   interfacing with underlying framework.
        /// </returns>
        TEntity Delete(TEntity entity);

        /// <summary>
        ///   Flushes the local cache of this repository instance.
        /// </summary>
        void Flush();

        /// <summary>
        ///   Gets a queryable which may be used to access all
        ///   entities contained in the underlying repository which
        ///   may not yet be cached locally.
        /// </summary>
        /// <returns>A queryable to the underlying data store.</returns>
        IQueryable<TEntity> GetAll();

        /// <summary>
        ///   Gets an entity by its id.
        /// </summary>
        /// <param name="id">The identifier.</param>
        /// <returns>
```

```
        ///   The entity with the given id (possibly a proxy instance)
        ///   or <see langword="null"/>.
        /// </returns>
        TEntity GetById(TKey id);

        /// <summary>
        ///   Gets the primary key from the given entity.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>The primary key of the given entity.</returns>
        TKey GetId(TEntity entity);

        /// <summary>
        ///   Adds the specified entity to this repository and places
        ///   it in queue for update when the parent data context has
        ///   changes on it accepted.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>
        ///   The entity given or a potential proxy instance for
        ///   interfacing with underlying framework.
        /// </returns>
        TEntity Update(TEntity entity);
    }
}
```

Listing C.11: src/Rel.Data/IRepository.cs

```
using System;

namespace Rel.Data
{
    /// <summary>
    ///   Identifies the various states of a lock.
    /// </summary>
    [Flags]
    public enum LockStatus
    {
        /// <summary>
        ///   No lock is held.
        /// </summary>
        Open = 0x00000000,

        /// <summary>
        ///   A lock is held.
        /// </summary>
        Closed = 0x00000001,

        /// <summary>
        ///   The current user holds the lock.
        /// </summary>
        Exclusive = 0x00000003
    }
}
```

Listing C.12: src/Rel.Data/LockStatus.cs

```
using Rel.Merge.Strategies;
using System.ComponentModel.DataAnnotations;

namespace Rel.Data.Models
{
    //[DirtyDelete]
    //[HiddenDelete]
    [LastWriteWins(false)]
    public class Asset
    {
        public int Id { get; set; }

        public int JobId { get; set; }

        //[DecaySpanMergeable("0.00:00:00.3", "0.00:00:00.3")]
        //[StepMergeable(true, 0.3)]
        //[LastWriteWins]
        public double? MaximumAndMinimumDecay { get; set; }

        //[DecaySpanMergeable("0.00:00:00.3", "0.00:00:00.3")]
        //[StepMergeable(true, 0.2)]
        public double? MaxMinDecayWithStepAndTol { get; set; }

        public double MinimumDecay { get; set; }

        [StepMergeable(0, 50, InclusiveLBound = false, InclusiveUBound = true)]
        public double? MonotonicTolerance { get; set; }

        //[LastWriteWins]
        public string Name { get; set; }

        [StepMergeable(true, 0.1, InclusiveLBound = true, InclusiveUBound = true)]
        public double? PercentTolerance { get; set; }

        [Timestamp]
        public byte[] RowVersion { get; set; }

        public string ServiceArea { get; set; }

        [StepMergeable(50, InclusiveLBound = true, InclusiveUBound = true)]
        public double? StaticTolerance { get; set; }
    }
}
```

Listing C.13: src/Rel.Data/Models/Asset.cs

```
using System;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;
```

```
namespace Rel.Data.Ef6
{
    /// <summary>
    ///   Exposes an EF DbSet as a
    ///   <see cref="T:Rel.Data.IRepository"/>. Queries directly on
    ///   DbRepository are *local only*.
    /// </summary>
    /// <typeparam name="TEntity">The type of the entity.</typeparam>
    /// <typeparam name="TKey">The type of the key.</typeparam>
    /// <remarks>
    ///   <note type="note">All direct queries access the local store,
    ///   not the traditional EF Linq-SQL provider.</note>
    ///   <para>
    ///     To access the EF Linq-SQL provider, use the
    ///     <see cref="M:GetAll()"/> method.
    ///   </para>
    /// </remarks>
    internal class DbRepository<TEntity, TKey>
        : IRepository<TEntity, TKey> where TEntity : class
    {
        private readonly Func<TEntity, TKey> _compiledKeySelector;
        private readonly DbSet<TEntity> _dbSet;
        private readonly Expression<Func<TEntity, TKey>> _keySelector;
        private readonly IQueryable<TEntity> _queryable;
        private DbContext _context;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="DbRepository{TEntity, TKey}"/> class.
        /// </summary>
        /// <param name="context">The context.</param>
        /// <param name="dbSet">The database set.</param>
        /// <param name="keySelector">The key selector.</param>
        /// <exception cref="System.ArgumentNullException">
        ///   context or dbSet or keySelector
        /// </exception>
        public DbRepository(DbContext context, DbSet<TEntity> dbSet, Expression<Func<
            TEntity, TKey>> keySelector)
        {
            if (context == null)
                throw new ArgumentNullException("context");
            if (dbSet == null)
                throw new ArgumentNullException("dbSet");
            if (keySelector == null)
                throw new ArgumentNullException("keySelector");

            _context = context;
            _dbSet = dbSet;
            _queryable = dbSet.Local.AsQueryable();
            _keySelector = keySelector;
            _compiledKeySelector = keySelector.Compile();
        }

        /// <summary>
        ///   Gets the type of the element(s) that are returned when
        ///   the expression tree associated with this instance of
        ///   <see cref="T:System.Linq.IQueryable"/> is executed.
        /// </summary>
        Type IQueryable.ElementType
        {
            get { return _queryable.ElementType; }
        }

        /// <summary>
        ///   Gets the expression tree that is associated with the
        ///   instance of <see cref="T:System.Linq.IQueryable"/>.
        /// </summary>
        Expression IQueryable.Expression
        {
            get { return _queryable.Expression; }
        }

        /// <summary>
        ///   Gets the query provider that is associated with this
        ///   data source.
        /// </summary>
        IQueryProvider IQueryable.Provider
        {
            get { return _queryable.Provider; }
        }

        /// <summary>
        ///   Gets an expression which, when compiled and run, yields
        ///   the primary key to entities of type TEntity.
        /// </summary>
        /// <value>The key selector.</value>
        Expression<Func<TEntity, TKey>> IRepository<TEntity, TKey>.KeySelector { get {
             return _keySelector; } }

        /// <summary>
        ///   Adds the specified entity to this repository and places
        ///   it in queue for creation when the parent data context
        ///   has changes on it accepted.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>
        ///   The entity given or a potential proxy instance for
        ///   interfacing with underlying framework.
        /// </returns>
        TEntity IRepository<TEntity, TKey>.Create(TEntity entity)
        {
            var e = _context.Entry(entity);
            e.State = EntityState.Added;
            return e.Entity;
        }

        /// <summary>
        ///   Adds the specified entity to this repository and places
        ///   it in queue for deletion when the parent data context
        ///   has changes on it accepted.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>
        ///   The entity given or a potential proxy instance for
        ///   interfacing with underlying framework.
        /// </returns>
        TEntity IRepository<TEntity, TKey>.Delete(TEntity entity)
        {
            var e = _context.Entry(entity);
            e.State = EntityState.Deleted;
            return e.Entity;
        }

        /// <summary>
        ///   Flushes the local cache of this repository instance.
        /// </summary>
        void IRepository<TEntity, TKey>.Flush()
        {
            foreach (var item in _dbSet.Local.ToArray())
            {
                _context.Entry(item).State = EntityState.Detached;
            }
        }

        /// <summary>
        ///   Gets a queryable which may be used to access all
        ///   entities contained in the underlying repository which
        ///   may not yet be cached locally.
        /// </summary>
        /// <returns>A queryable to the underlying data store.</returns>
        IQueryable<TEntity> IRepository<TEntity, TKey>.GetAll()
        {
            return _dbSet;
        }

        /// <summary>
        ///   Gets an entity by its id.
        /// </summary>
        /// <param name="id">The identifier.</param>
        /// <returns>
        ///   The entity with the given id (possibly a proxy instance)
        ///   or <see langword="null"/>.
        /// </returns>
        TEntity IRepository<TEntity, TKey>.GetById(TKey id)
        {
            var selector = BuildSelectorFor(id);
            TEntity result;
            try
            {
                result = _dbSet.Local.AsQueryable().SingleOrDefault(selector);
                result = result ?? _dbSet.SingleOrDefault(selector);
                return result;
            }
            catch (InvalidOperationException)
            {
                System.Diagnostics.Trace.TraceError("Caught duplicate exception id",
                    id);
                throw;
            }
        }

        /// <summary>
        ///   Gets the primary key from the given entity.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>The primary key of the given entity.</returns>
        TKey IRepository<TEntity, TKey>.GetId(TEntity entity)
        {
            return _compiledKeySelector(entity);
        }

        /// <summary>
        ///   Adds the specified entity to this repository and places
        ///   it in queue for update when the parent data context has
        ///   changes on it accepted.
        /// </summary>
        /// <param name="entity">The entity.</param>
        /// <returns>
        ///   The entity given or a potential proxy instance for
        ///   interfacing with underlying framework.
        /// </returns>
        TEntity IRepository<TEntity, TKey>.Update(TEntity entity)
        {
            var e = _context.Entry(entity);
            e.State = EntityState.Modified;

            return e.Entity;
        }

        /// <summary>
        ///   Returns an enumerator that iterates through the collection.
        /// </summary>
        /// <returns>
        ///   A
        ///   <see cref="T:System.Collections.Generic.IEnumerator`1"/>
        ///   that can be used to iterate through the collection.
        /// </returns>
        System.Collections.Generic.IEnumerator<TEntity> System.Collections.Generic.
            IEnumerable<TEntity>.GetEnumerator()
        {
            return _queryable.GetEnumerator();
        }

        /// <summary>
        ///   Returns an enumerator that iterates through a collection.
        /// </summary>
        /// <returns>
        ///   An <see cref="T:System.Collections.IEnumerator"/> object
        ///   that can be used to iterate through the collection.
        /// </returns>
        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return _queryable.GetEnumerator();
```

```
        }

        /// <summary>
        ///   Builds the selector for.
        /// </summary>
        /// <param name="id">The identifier.</param>
        /// <returns></returns>
        private Expression<Func<TEntity, bool>> BuildSelectorFor(TKey id)
        {
            var eq = Expression.Equal(_keySelector.Body, Expression.Constant(id));
            return Expression.Lambda<Func<TEntity, bool>>(eq, _keySelector.Parameters)
                ;
        }
    }
}
```

Listing C.14: src/Rel.Data.Ef6/DbRepository.cs

```
using Rel.Data.Models;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;

namespace Rel.Data.Ef6
{
    /// <summary>
    ///   Exposes an EF6 <see cref="T:System.Data.Entity.DbContext"/>
    ///   as a <see cref="T:Rel.Data.IDataContext"/> to allow for
    ///   simple replacement of the underlying data persistence layer.
    /// </summary>
    public class TpContext
        : DbContext, IDataContext
    {
        private readonly DbRepository<Asset, int> _assets;
        private readonly DbRepository<Job, int> _jobs;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="TpContext"/> class.
        /// </summary>
        public TpContext()
            : base("TpContext")
        {
            Configuration.LazyLoadingEnabled = false;

            _assets = new DbRepository<Asset, int>(this, Assets, _ => _.Id);
            _jobs = new DbRepository<Job, int>(this, Jobs, _ => _.Id);
        }

        /// <summary>
        ///   Gets the asset repository.
        /// </summary>
        /// <value>The assets.</value>
        public DbSet<Asset> Assets { get; set; }
```

```
/// <summary>
///   Gets the asset repository.
/// </summary>
/// <value>The assets.</value>
IRepository<Asset, int> IDataContext.Assets
{
    get { return _assets; }
}

/// <summary>
///   Gets the job repository.
/// </summary>
/// <value>The jobs.</value>
IRepository<Job, int> IDataContext.Jobs
{
    get { return _jobs; }
}

/// <summary>
///   Gets the job repository.
/// </summary>
/// <value>The jobs.</value>
public DbSet<Job> Jobs { get; set; }

/// <summary>
///   <para>
///     Attempts to push any changes made in this data context
///     since the last time it communicated with the
///     underlying data store to persisted storage.
///   </para>
///   <para>
///     Any error preventing a complete commit of the changes
///     within throws an exception.
///   </para>
/// </summary>
void IDataContext.AcceptChanges()
{
    SaveChanges();
}

/// <summary>
///   Undoes any changes made in this data context since the
///   last time it communicated with the underlying data store.
/// </summary>
void IDataContext.RejectChanges()
{
    this.RejectChanges();
}

/// <summary>
///   Validates changes in this data context.
/// </summary>
/// <returns>
///   <see langword="true"/> if valid; otherwise, false.
/// </returns>
```

```
bool IDataContext.Validate()
{
    return !GetValidationErrors().Any();
}

/// <summary>
///   Saves all changes made in this context to the underlying database.
/// </summary>
/// <returns>
///   The number of state entries written to the underlying
///   database. This can include state entries for entities
///   and/or relationships. Relationship state entries are
///   created for many-to-many relationships and relationships
///   where there is no foreign key property included in the
///   entity class (often referred to as independent associations).
/// </returns>
/// <exception cref="ConcurrencyException">
///   Dirty writes detected.
/// </exception>
public override int SaveChanges()
{
    var errors = GetValidationErrors();
    if (!errors.Any())
    {
        try
        {
            return base.SaveChanges();
        }
        catch (System.Data.Entity.Infrastructure.DbUpdateConcurrencyException
            ex)
        {
            throw new ConcurrencyException("Dirty writes detected.", ex);
        }
    }
    else
    {
        throw new ValidationException();
    }
}

/// <summary>
///   Calls the protected Dispose method.
/// </summary>
void System.IDisposable.Dispose()
{
    this.Dispose();
}

/// <summary>
///   This method is called when the model for a derived
///   context has been initialized, but before the model has
///   been locked down and used to initialize the context. The
///   default implementation of this method does nothing, but
///   it can be overridden in a derived class such that the
///   model can be further configured before it is locked down.
/// </summary>
```

```
        /// <param name="modelBuilder">
        ///   The builder that defines the model for the context being created.
        /// </param>
        /// <remarks>
        ///   Typically, this method is called only once when the
        ///   first instance of a derived context is created. The
        ///   model for that context is then cached and is for all
        ///   further instances of the context in the app domain. This
        ///   caching can be disabled by setting the ModelCaching
        ///   property on the given ModelBuidler, but note that this
        ///   can seriously degrade performance. More control over
        ///   caching is provided through use of the DbModelBuilder
        ///   and DbContextFactory classes directly.
        /// </remarks>
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
            modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>();

            modelBuilder
                .ConfigureJobs()
                .ConfigureAssets();
        }

        /// <summary>
        ///   Undoes any changes made in this data context since the
        ///   last time it communicated with the underlying data store.
        /// </summary>
        private void RejectChanges()
        {
            foreach (var entry in ChangeTracker.Entries())
            {
                if (entry.State == EntityState.Added)
                {
                    entry.State = EntityState.Detached;
                }
                else
                {
                    entry.State = EntityState.Unchanged;
                }
            }
        }
    }
}
```

Listing C.15: src/Rel.Data.Ef6/TpContext.cs

C.3 Merge

```
namespace Rel.Merge
{
    /// <summary>
    ///   Stubs the core purpose of IMergeResolution{} for internal
    ///   communication and state sharing.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    internal class BasicMergeResolution<T>
        : IMergeResolution<T>
    {
        private MergeActionResult _result;
        private T _value;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="BasicMergeResolution{T}"/> class.
        /// </summary>
        /// <param name="result">The result.</param>
        /// <param name="value">The value.</param>
        public BasicMergeResolution(MergeActionResult result, T value)
        {
            _result = result;
            _value = value;
        }

        /// <summary>
        ///   Gets the resolved value.
        /// </summary>
        /// <value>The resolved value.</value>
        public T ResolvedValue
        {
            get { return _value; }
        }

        /// <summary>
        ///   Gets the result.
        /// </summary>
        /// <value>The result.</value>
        public MergeActionResult Result
        {
            get { return _result; }
        }
    }
}
```

Listing C.16: src/Rel.Merge/BasicMergeResolution.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Rel.Merge
{
    /// <summary>
    ///   Identifies an object which can merge arbitrary objects with
    ///   minimal boxing.
    /// </summary>
    public interface IMergeProvider
    {
        /// <summary>
        ///   Merges the given entities understanding the intent of the
        ///   merge by the kind specified.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <param name="kind">The kind of merge.</param>
        /// <param name="before">The before image.</param>
        /// <param name="current">The current image.</param>
        /// <param name="after">The after image.</param>
        /// <returns>A resolution for the merge request.</returns>
        IMergeResolution<TEntity> Merge<TEntity>(MergeKind kind, TEntity before,
            TEntity current, TEntity after);
    }
}
```

Listing C.17: src/Rel.Merge/IMergeProvider.cs

```
using System;

namespace Rel.Merge
{
    /// <summary>
    ///   Identifies the result a merge operation.
    /// </summary>
    /// <typeparam name="TResolved">The type of the resolved.</typeparam>
    public interface IMergeResolution<TResolved>
    {
        /// <summary>
        ///   Gets the resolved value.
        /// </summary>
        /// <value>The resolved value.</value>
        TResolved ResolvedValue { get; }

        /// <summary>
        ///   Gets the result.
        /// </summary>
        /// <value>The result.</value>
        MergeActionResult Result { get; }
    }

    /// <summary>
    ///   Simple convenience helpers to avoid having to implement on any
        IMergeResolution{}.
    /// </summary>
    public static class MergeResolutionExtension
    {
        /// <summary>
        ///   Determines whether this instance is resolved.
        /// </summary>
        /// <typeparam name="TResolved">The type of the resolved.</typeparam>
        /// <param name="resolution">The resolution.</param>
        /// <returns>
        ///   <see langword="true"/> if this instance has resolved;
        ///   otherwise, <see langword="false"/>.
        /// </returns>
        public static bool IsResolved<TResolved>(this IMergeResolution<TResolved>
            resolution)
        {
            return resolution.Result.HasFlag(MergeActionResult.Resolved);
        }
    }
}
```

Listing C.18: src/Rel.Merge/IMergeResolution.cs

```
using System;

namespace Rel.Merge
{
    /// <summary>
    ///   Communicates steps in the merge process to individual merge implementations.
    /// </summary>
    /// <typeparam name="TValue">The type of the value.</typeparam>
    public class MergeAction<TValue>
    {
        private readonly TValue _bfim, _cfim, _afim;
        private readonly MergeKind _kind;
        private TValue _resolvedValue;
        private MergeActionResult? _result;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="MergeAction{TValue}"/> class.
        /// </summary>
        /// <param name="kind">The kind of merge.</param>
        /// <param name="bfim">
        ///   The before image, shared base state between cfim and afim.
        /// </param>
        /// <param name="cfim">The current image.</param>
        /// <param name="afim">
        ///   The after image attempting to overwrite cfim.
        /// </param>
        public MergeAction(MergeKind kind, TValue bfim, TValue cfim, TValue afim)
        {
            _kind = kind;
            _bfim = bfim;
            _cfim = cfim;
            _afim = afim;
            _resolvedValue = default(TValue);
            _result = null;
        }

        /// <summary>
        ///   Gets the after image.
        /// </summary>
        /// <value>The after image.</value>
        public TValue AFIM { get { return _afim; } }
        /// <summary>
        ///   Gets the before image.
        /// </summary>
        /// <value>The before image.</value>
        public TValue BFIM { get { return _bfim; } }

        /// <summary>
        ///   Gets the current image.
        /// </summary>
        /// <value>The current image.</value>
        public TValue CFIM { get { return _cfim; } }

        /// <summary>
        ///   Gets the kind of merge.
        /// </summary>
        /// <value>The kind of merge.</value>
        public MergeKind Kind { get { return _kind; } }

        /// <summary>
        ///   Gets a value indicating whether this
        ///   <see cref="MergeAction{TValue}"/> is resolved.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if resolved; otherwise, <see langword="false"/>.
        /// </value>
        public bool Resolved
        {
            get
            {
                return _result.HasValue &&
_result.Value.HasFlag(MergeActionResult.Resolved);
            }
        }

        /// <summary>
        ///   Gets the value this action resolved to.
        /// </summary>
        /// <value>The resolved value.</value>
        public TValue ResolvedValue { get { return _resolvedValue; } }

        /// <summary>
        ///   Gets the result an attempt to merge on this action.
        /// </summary>
        /// <value>The result.</value>
        public MergeActionResult Result
        {
            get
            {
                return
                    _result ??
                    MergeActionResult.Unresolved;
            }
        }

        /// <summary>
        ///   Resolves using the specified result and value.
        /// </summary>
        /// <param name="result">The result.</param>
        /// <param name="resolveWith">The resolve with.</param>
        /// <exception cref="System.ArgumentException">
        ///   Cannot resolve with Unresolved.;result
        /// </exception>
        /// <exception cref="System.InvalidOperationException">
        ///   Already resolved.
        /// </exception>
        public void Resolve(MergeActionResult result, TValue resolveWith)
        {
            if (!result.HasFlag(MergeActionResult.Resolved))
                throw new ArgumentException("Cannot resolve with Unresolved.", "result
                    ");

            if (_result.HasValue)
                throw new InvalidOperationException("Already resolved.");

            _result = result;
            _resolvedValue = resolveWith;
        }
    }
}
```

Listing C.19: src/Rel.Merge/MergeAction.cs

```
using System;

namespace Rel.Merge
{
    /// <summary>
    ///   Identifies action taken in response to a merge operation.
    /// </summary>
    [Flags]
    public enum MergeActionResult
    {
        /// <summary>
        ///   No action taken, merge cannot be resolved.
        /// </summary>
        Unresolved = 0x00000000,

        /// <summary>
        ///   Identifies that the action was resolved. If no sibling
        ///   accent states appear with this state, the merge is
        ///   resolve through NOOP.
        /// </summary>
        Resolved = 0x00000001,

        /// <summary>
        ///   Merge is resolved by deletion.
        /// </summary>
        Delete = 0x00000003,

        /// <summary>
        ///   Merge is resolved by creation.
        /// </summary>
        Create = 0x00000005,

        /// <summary>
        ///   Merge is resolved by update to existing state.
        /// </summary>
        Update = 0x00000009
    }
}
```

Listing C.20: src/Rel.Merge/MergeActionResult.cs

```
namespace Rel.Merge
{
    /// <summary>
    ///   Identify the reason for invocation of a merge action.
    /// </summary>
    public enum MergeKind
    {
        /// <summary>
        ///   Let merge decide the kind of operation based upon
        ///   bfim,cfim,and afim.
        /// </summary>
        Auto = 0,

        /// <summary>
        ///   The action is attempting to resolve a conflicting update.
        /// </summary>
        ConflictingUpdate,

        /// <summary>
        ///   The action is attempting to reconcile a new write
        ///   (delete or update) with a delete which has previously
        ///   been committed.
        /// </summary>
        HiddenDelete,

        /// <summary>
        ///   The action is attempting to resolve a previously
        ///   committed change with a new delete action.
        /// </summary>
        DirtyDelete
    }
}
```

Listing C.21: src/Rel.Merge/MergeKind.cs

```
 using Rel.Merge.Strategies;
using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
```

```
namespace Rel.Merge
{
    /// <summary>
    ///   Repents a merge operation.
    /// </summary>
    public class MergeOperation : IMergeProvider
    {
        /// <summary>
        ///   Performs a merge of the specified kind.
        /// </summary>
        /// <typeparam name="TEntity">The type of the entity.</typeparam>
        /// <param name="kind">The kind.</param>
        /// <param name="before">The before.</param>
        /// <param name="current">The current.</param>
        /// <param name="after">The after.</param>
        /// <returns></returns>
        public IMergeResolution<TEntity> Merge<TEntity>(MergeKind kind, TEntity before
            , TEntity current, TEntity after)
        {
            MergeOperation<TEntity> op;

            op = new MergeOperation<TEntity>(before, current, after);

            var result = op.Merge();
            bool resolved = result.IsResolved();
            return result;
        }
    }

    /// <summary>
    ///   Encapsulates the merge API behind a concrete type which
    ///   caches merge resolution strategies by the entity type.
    /// </summary>
    /// <typeparam name="TEntity">The type of the entity.</typeparam>
    internal class MergeOperation<TEntity>
    {
        private static readonly CloneMethod[] s_cloneProps;

        /// <summary>
        ///   Tests that the BFIM of a conflict is current with the
        ///   Current state.
        /// </summary>
        private static readonly CurrentStateCheck s_isCurrent;

        /// <summary>
        ///   The dynamic merge implementation used.
        /// </summary>
        private static readonly Func<MergeOperation<TEntity>, IMergeResolution<TEntity
            >> s_mergeImpl;

        /// <summary>
        ///   Invokes the merge on all properties which are not
        ///   control properties.
        /// </summary>
        private static readonly PropertyMergeMethod[] s_propertyMerges;
        private static readonly MergeableAttribute[] s_typeAttr;
        private TEntity _afim;
        private TEntity _bfim;
        private TEntity _current;

        /// <summary>
        ///   Initializes the <see cref="MergeOperation{TEntity}"/> class.
        /// </summary>
        static MergeOperation()
        {
            var type = typeof(TEntity);
            var props = type.GetProperties();

            if (InitCc(type, props, out s_mergeImpl, ref s_isCurrent))
            {
                InitMerge(type, props, ref s_propertyMerges, ref s_typeAttr, ref
                    s_cloneProps);
            }
        }

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="MergeOperation{TEntity}"/> class.
        /// </summary>
        /// <param name="before">The before image.</param>
        /// <param name="current">The current image.</param>
        /// <param name="after">The after image.</param>
        public MergeOperation(TEntity before, TEntity current, TEntity after)
        {
            _bfim = before;
            _current = current;
            _afim = after;
        }

        /// <summary>
        ///   Clones data from src to dest.
        /// </summary>
        /// <param name="src">The source.</param>
        /// <param name="dest">The destination.</param>
        private delegate void CloneMethod(TEntity src, TEntity dest);

        /// <summary>
        ///   Signature of concurrency checking implementations.
        /// </summary>
        /// <param name="bfim">The before image.</param>
        /// <param name="current">The current image.</param>
        /// <returns></returns>
        private delegate bool CurrentStateCheck(TEntity bfim, TEntity current);

        /// <summary>
        ///   Signature of property level merging.
        /// </summary>
        /// <param name="before">The before image.</param>
        /// <param name="current">The current image.</param>
        /// <param name="after">The after image.</param>
        /// <returns></returns>
```

```
private delegate PendingMergeResolution PropertyMergeMethod(TEntity before,
    TEntity current, TEntity after);

/// <summary>
///   Signature for class level merging.
/// </summary>
/// <param name="kind">The kind of merge resolved by <see cref="M:
    ControlledMerge"/>.</param>
/// <param name="mop">The merge operation.</param>
/// <returns></returns>
private delegate IMergeResolution<TEntity> TypeMergeMethod(MergeKind kind,
    MergeOperation<TEntity> mop);

/// <summary>
///   Gets the final after image resultant from this merge operation.
/// </summary>
/// <value>The final after image value..</value>
public TEntity AFIM { get { return _afim; } }

/// <summary>
///   Determines if two OCCUTS values match.
/// </summary>
/// <param name="cc1">The CC1.</param>
/// <param name="cc2">The CC2.</param>
/// <returns>
///   <see langword="true"/> if the time stamps match;
///   otherwise, <see langword="false"/>.
/// </returns>
internal static bool TimestampIsCurrentChecker(byte[] cc1, byte[] cc2)
{
    if (cc1 == null)
    {
        return cc2 == null;
    }
    else if (cc2 == null)
    {
        return cc1 == null;
    }
    else if (object.ReferenceEquals(cc1, cc2))
    {
        return true;
    }
    else if (cc1.Length != cc2.Length)
    {
        return false;
    }
    else if (cc1.Length == 8)
    {
        // unrolled loop evaluation for performance as this
        // method should see a lot of mileage
        return
            cc1[0] == cc2[0] &&
            cc1[1] == cc2[1] &&
            cc1[2] == cc2[2] &&
            cc1[3] == cc2[3] &&
            cc1[4] == cc2[4] &&
            cc1[5] == cc2[5] &&
            cc1[6] == cc2[6] &&
            cc1[7] == cc2[7];
    }
    else
    {
        return cc1.SequenceEqual(cc2);
    }
}

/// <summary>
///   Performs the merge operation.
/// </summary>
internal IMergeResolution<TEntity> Merge()
{
    return s_mergeImpl(this);
}

/// <summary>
///   Effectively last write wins. "Merge" where TEntity does
///   not participate in optimistic concurrency control.
/// </summary>
/// <returns><see langword="true"/> always.</returns>
/// <remarks>
///   An entity which does not participate in concurrency
///   control is implicitly last write wins which mirrors
///   regular RDBMS behavior.
/// </remarks>
private static IMergeResolution<TEntity> ChaoticMerge(MergeOperation<TEntity>
    op)
{
    return Resolve(MergeActionResult.Resolved, op.AFIM);
}

/// <summary>
///   Clones the non-control properties.
/// </summary>
/// <param name="src">The source.</param>
/// <param name="dest">The destination.</param>
private static void CloneNonControl(TEntity src, TEntity dest)
{
    for (int i = s_cloneProps.Length; i-- > 0; )
    {
        s_cloneProps[i](src, dest);
    }
}

/// <summary>
///   The concurrency controlled merge implementation.
/// </summary>
/// <param name="op">The merge operation invoking merge.</param>
/// <returns>
///   <see langword="true"/> if merge was successful;
///   otherwise, <see langword="false"/>.
/// </returns>
```

```
private static IMergeResolution<TEntity> ControlledMerge(MergeOperation<
    TEntity> op)
{
    var bfim = op._bfim;

    // should we already be aware of it? no? this is new,
    // implies no conflict is possible (in this version at least)
    if (bfim == null)
        return Resolve(MergeActionResult.Create, op.AFIM);

    var cfim = op._current;

    // did someone else already delete this while bfim was
    // offline? yes? handle as special case since property
    // merge strategies have no basis for comparison
    if (cfim == null)
        return MergeByType(MergeKind.HiddenDelete, op);

    // is this normal optimistic write behavior? yes? then why
    // waste time fancy dancing?
    if (s_isCurrent(bfim, cfim))
        return Resolve(MergeActionResult.Resolved, op._afim);

    var afim = op._afim;

    // are we trying to delete someone else's work? yes?
    // handle as special case since the afim cannot be used
    // for comparison and the other writer presumably worked
    // from bfim to get to current.
    if (afim == null)
        return MergeByType(MergeKind.DirtyDelete, op);

    // finally, we have work todo. This is a legitimate
    // conflicting update
    return MergeByType(MergeKind.ConflictingUpdate, op);
}

/// <summary>
///   Creates a pre-compiled wrapper to clone data between two entities.
/// </summary>
/// <param name="property">The property.</param>
/// <returns>
///   A precompiled method which will clone values from one
///   entity to another for the property given.
/// </returns>
private static CloneMethod CreateCloneWrapper(PropertyInfo property)
{
    var type = typeof(TEntity);
    var src = Expression.Parameter(typeof(TEntity));
    var dest = Expression.Parameter(typeof(TEntity));

    var expr = Expression
        .Lambda<CloneMethod>(
            Expression.Call(
                dest,
                property.GetSetMethod(),
                Expression.Call(
                    src,
                    property.GetGetMethod())
            ),
        src, dest);

    return expr.Compile();
}

/// <summary>
///   Creates a precompiled wrapper to evaluate whether two
///   entities share the same timestamp value for optimistic
///   concurrency control.
/// </summary>
/// <param name="ccProperty">The cc property.</param>
/// <returns></returns>
/// <exception cref="MergeException">
///   Timestamp field must be readable. or Timestamp
///   attributed fields must be of type byte[].
/// </exception>
private static CurrentStateCheck CreateOccUtsCurrencyChecker(PropertyInfo
    ccProperty)
{
    // current version only support Timestamp concurrency
    // evaluation. A future revision may support the
    // System.ComponentModel.DataAnnotations.ConcurrencyCheckAttribute
    // as well.

    if (!ccProperty.CanRead)
    {
        throw new MergeException("Timestamp field must be readable.");
    }

    if (ccProperty.PropertyType != typeof(byte[]))
    {
        throw new MergeException("Timestamp attributed fields must be of type
            byte[].");
    }

    var fGet = PropertyWrapper.CreatePropertyGetter<TEntity, byte[]>(
        ccProperty);
    return (TEntity a, TEntity b) => TimestampIsCurrentChecker(fGet(a), fGet(b
        ));
}

/// <summary>
///   Initializes the concurrency control domain for the given type.
/// </summary>
/// <param name="type">The type.</param>
/// <param name="properties">The properties.</param>
/// <param name="merge">The merge.</param>
/// <param name="IsCurrent">The is current.</param>
/// <returns>
///   <see langword="true"/> if the domain is controlled;
///   otherwise, <see langword="false"/>.
/// </returns>
/// <exception cref="System.InvalidOperationException">
///   Only one timestamp property is permitted per type.
/// </exception>
private static bool InitCc(Type type, PropertyInfo[] properties, out Func<
    MergeOperation<TEntity>, IMergeResolution<TEntity>> merge, ref
    CurrentStateCheck IsCurrent)
{
    // current version only supports Timestamp concurrency
    // evaluation. A future revision may support the
    // System.ComponentModel.DataAnnotations.ConcurrencyCheckAttribute
    // as well.
    PropertyInfo ccProperty;

    try
    {
        // in a future revision, extend this with inspection
        // for
        // System.ComponentModel.DataAnnotations.ConcurrencyCheckAttribute
        // and return a method which looks like (a,b)=>AllCcProps.All(_=>_(a,b
            ));
        ccProperty = properties
            .Where(_ => _.GetCustomAttributes(true).OfType<System.
                ComponentModel.DataAnnotations.TimestampAttribute>().Any())
            .SingleOrDefault();
    }
    catch (InvalidOperationException)
    {
        throw new InvalidOperationException("Only one timestamp property is
            permitted per type.");
    }

    // in the absence CC, there exists no possibility of
    // conflict / detection.
    if (ccProperty == null)
    {
        merge = ChaoticMerge;
        return false;
    }

    IsCurrent = CreateOccUtsCurrencyChecker(ccProperty);
    merge = ControlledMerge;

    return true;
}

/// <summary>
///   Initializes the merge domain of this class.
/// </summary>
/// <param name="type">The type to merge on.</param>
/// <param name="properties">
///   The properties of the given type.
/// </param>
/// <param name="propertyMerges">
///   A collection of property wrappers which will perform the
///   merge of data from one entity to the next.
/// </param>
/// <param name="typeMergeAttr">
///   The mergeable attributes applied to the type given.
/// </param>
/// <param name="cloneProps">
///   A collection of property wrappers to move data between
///   entities for all but control fields.
/// </param>
private static void InitMerge(Type type, PropertyInfo[] properties,
    ref PropertyMergeMethod[] propertyMerges,
    ref MergeableAttribute[] typeMergeAttr,
    ref CloneMethod[] cloneProps)
{
    typeMergeAttr = type.GetCustomAttributes<MergeableAttribute>(false).
        ToArray();

    var controlProps = properties.Where(IsControlProperty).ToArray();
    var mergedProps = properties.Where(IsMergedProperty).ToArray();
    var unmergedProps = properties
        .Except(controlProps)
        .Except(mergedProps).ToArray();

    LastWriteWinsAttribute unmergedPropertyDefault = null;
    if (typeMergeAttr.Length == 0)
    {
        // type defaults to reject
        typeMergeAttr = new[] { new LastWriteWinsAttribute(false) };

        unmergedPropertyDefault = new LastWriteWinsAttribute(mergedProps.
            Length > 0);
    }
    else
    {
        unmergedPropertyDefault = new LastWriteWinsAttribute(false);
    }

    if (mergedProps.Length == 0)
    {
        propertyMerges = new PropertyMergeMethod[] { NeverMerge };
    }
    else
    {
        var p = mergedProps
            .Select(_ => WrapProperty(_, _.GetCustomAttributes<
                MergeableAttribute>(false).ToArray()))
            .Union(unmergedProps.Select(_ => WrapProperty(_,
                unmergedPropertyDefault)))
            .ToArray();

        propertyMerges = p.ToArray();
    }

    cloneProps = properties.Except(controlProps)
        .Select(CreateCloneWrapper)
        .ToArray();
}
```

```
/// <summary>
///   Determines whether the property given is a control
///   property. That is, a key field or a concurrency control field.
/// </summary>
/// <param name="property">The property.</param>
/// <returns>
///   <see langword="true"/> if the given property is a
///   controlled field.
/// </returns>
private static bool IsControlProperty(PropertyInfo property)
{
    var controlMembers = property
        .GetCustomAttributes(typeof(KeyAttribute), true)
        .Union(
        property.GetCustomAttributes<ConcurrencyCheckAttribute>(true))
        .Union(
        property.GetCustomAttributes<TimestampAttribute>(true));

    return controlMembers.Any();
}

/// <summary>
///   Determines whether the given propert is a merged
///   property. That is, will participate in verifying the
///   validity of data resulting from dirty writes.
/// </summary>
/// <param name="property">The property.</param>
/// <returns>
///   <see langword="true"/> if the property participates in
///   merge; otherwise, <see langword="false"/>.
/// </returns>
private static bool IsMergedProperty(PropertyInfo property)
{
    if (!property.GetCustomAttributes<MergeableAttribute>(false).Any())
        return false;
    if (IsControlProperty(property))
    {
        System.Diagnostics.Trace.TraceWarning("Merge attributes on property
            {0} of type {1} will be ignored due to control attributes applied.
            ",
            property.Name, property.DeclaringType.FullName);
        return false;
    }
    return true;
}

/// <summary>
///   Merges all mergeable properties.
/// </summary>
/// <param name="kind">
///   The kind of merge operation; only here for signature compliance..
/// </param>
/// <param name="op">The merge operation.</param>
/// <returns>
///   A merge resolution which reports as resolved if all
///   merged properties could be merged; otherwise, unresolvable.
/// </returns>
private static IMergeResolution<TEntity> MergeAllProperties(MergeKind kind,
    MergeOperation<TEntity> op)
{
    // the result are cached and processed after so that the
    // parent type merge can process clean data should the
    // property merge fail.
    var results = new PendingMergeResolution[s_propertyMerges.Length];
    for (int i = results.Length; i-- > 0; )
    {
        var result = s_propertyMerges[i](op._bfim, op._current, op._afim);
        if (!result.CanResolve)
            return Resolve(MergeActionResult.Unresolved, op._current);
        results[i] = result;
    }

    for (int i = results.Length; i-- > 0; )
    {
        results[i].Commit();
    }
    return Resolve(MergeActionResult.Update, op._current);
}

/// <summary>
///   Entry into merge on the given entity.
/// </summary>
/// <param name="mergeKind">Kind of the merge.</param>
/// <param name="op">The merge operation.</param>
/// <returns>
///   A merge resolution which reports as resolved if all
///   merged properties could be merged; otherwise, unresolvable.
/// </returns>
private static IMergeResolution<TEntity> MergeByType(MergeKind mergeKind,
    MergeOperation<TEntity> op)
{
    if (mergeKind == MergeKind.ConflictingUpdate)
    {
        // finally, we've work to do! This is a dirty update
        // request and does warrant merging. start with the
        // most fine grain merge possible
        var test = MergeAllProperties(MergeKind.ConflictingUpdate, op);

        if (test.IsResolved())
        {
            return test;
        }
    }

    var resolution = MergeType(mergeKind, op);
    var isResolved = resolution.IsResolved();
    return resolution;
}

/// <summary>
///   Performs the actual type level merge.
/// </summary>
/// <param name="kind">The kind.</param>
/// <param name="mop">The mop.</param>
/// <returns>
///   A merge resolution which reports as resolved if all
///   merged properties could be merged; otherwise, unresolvable.
/// </returns>
private static IMergeResolution<TEntity> MergeType(MergeKind kind,
    MergeOperation<TEntity> mop)
{
    TEntity result = default(TEntity);
    var action = new MergeAction<TEntity>(kind, mop._bfim, mop._current, mop.
        _afim);
    for (int i = s_typeAttr.Length; i-- > 0; )
    {
        s_typeAttr[i].Merge(action);
        if (action.Resolved)
        {
            if (mop._current != null && !object.ReferenceEquals(action.
                ResolvedValue, mop._current))
            {
                CloneNonControl(mop._afim, mop._current);
                result = mop._current;
            }
            else
            {
                result = mop._afim;
            }
            break;
        }
    }

    return Resolve(action.Result, result);
}

/// <summary>
///   A property merge implementation which always rejects.
/// </summary>
/// <param name="bfim">The bfim.</param>
/// <param name="current">The current.</param>
/// <param name="afim">The afim.</param>
/// <returns><see langword="false"/></returns>
private static PendingMergeResolution NeverMerge(TEntity bfim, TEntity current
    , TEntity afim)
{
    return new PendingMergeResolution(MergeActionResult.Unresolved, Noop);
}

/// <summary>
///   Convenience no operation method.
/// </summary>
private static void Noop()
{
}

/// <summary>
///   Produces an IMergeResolution{} using the resolution and
```

```
        ///   value given.
        /// </summary>
        /// <param name="result">The result.</param>
        /// <param name="tEntity">The t entity.</param>
        /// <returns>A completed merge resoution.</returns>
        private static IMergeResolution<TEntity> Resolve(MergeActionResult result,
            TEntity tEntity)
        {
            return new BasicMergeResolution<TEntity>(result, tEntity);
        }

        /// <summary>
        ///   Wraps the specified property with a merge acceptor
        /// </summary>
        /// <param name="prop">The property.</param>
        /// <param name="attrs">The merge attributes.</param>
        /// <returns>Accessor methods to the property merge.</returns>
        /// <exception cref="System.ArgumentException">
        ///   attrs must contain at least one mergeable attribute to wrap.
        /// </exception>
        private static PropertyMergeMethod WrapProperty(PropertyInfo prop, params
            MergeableAttribute[] attrs)
        {
            if (attrs == null || attrs.Length == 0)
                throw new ArgumentException("attrs must contain at least one mergeable
                     attribute to wrap.");

            var w = typeof(MergePropertyWrapper<,>).MakeGenericType(prop.DeclaringType
                , prop.PropertyType);
            var wrapper = Activator.CreateInstance(w, new object[] { prop, attrs },
                null);
            var kind = Expression.Constant(MergeKind.ConflictingUpdate); //Expression.
                Parameter(typeof(MergeKind));
            var b = Expression.Parameter(prop.DeclaringType);
            var c = Expression.Parameter(prop.DeclaringType);
            var a = Expression.Parameter(prop.DeclaringType);

            var exec = Expression.Lambda<PropertyMergeMethod>(
                Expression.Call(
                Expression.Constant(wrapper),
                w.GetMethod("Merge"),
                kind, b, c, a),
                /*kind,*/ b, c, a);

            return exec.Compile();
        }
    }
}
```

Listing C.22: src/Rel.Merge/MergeOperation.cs

```
using System;
using System.Linq;
using System.Reflection;
```

```
namespace Rel.Merge
{
    /// <summary>
    ///   A wrapper for properties which are mergeable.
    /// </summary>
    /// <typeparam name="TEntity">The type of the entity.</typeparam>
    /// <typeparam name="TProperty">The type of the property.</typeparam>
    internal class MergePropertyWrapper<TEntity, TProperty>
        : PropertyWrapper<TEntity, TProperty>
    {
        private Strategies.MergeableAttribute[] mattrs;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="MergePropertyWrapper{TEntity, TProperty}"/> class.
        /// </summary>
        /// <param name="info">The information.</param>
        /// <param name="mattrs">The merge attributes.</param>
        public MergePropertyWrapper(PropertyInfo info,
            Strategies.MergeableAttribute[] mattrs)
            : base(info)
        {
            this.mattrs = mattrs;
        }

        /// <summary>
        ///   Merges the specified entities.
        /// </summary>
        /// <param name="kind">
        ///   The kind of merge operation being attempted. Here for
        ///   signature compliance.
        /// </param>
        /// <param name="baseValue">The base value.</param>
        /// <param name="current">The current.</param>
        /// <param name="modifiedValue">The modified value.</param>
        /// <returns>An intermediate result with a commit callback.</returns>
        public PendingMergeResolution Merge(MergeKind kind,
            TEntity baseValue,
            TEntity current,
            TEntity modifiedValue)
        {
            TProperty
                b = Get(baseValue),
                c = Get(current),
                n = Get(modifiedValue);

            var action = new MergeAction<TProperty>(kind, b, c, n);

            if (current == null)
                current = modifiedValue;

            mattrs.FirstOrDefault(_ =>
            {
                _.Merge(action);
                return action.Resolved;
            });
            if (action.Resolved)
                return new PendingMergeResolution(action.Result,
                    new Action(() => Set(current, Get(modifiedValue))));

            return new PendingMergeResolution(action.Result, Noop);
        }

        /// <summary>
        ///   Convenience no operation method.
        /// </summary>
        private static void Noop()
        {
        }
    }
}
```

Listing C.23: src/Rel.Merge/MergePropertyWrapper.cs

```
using System;

namespace Rel.Merge
{
    /// <summary>
    ///   ***internal use only*** An intermediate result which
    ///      communicates state of child merge operations with a
    ///      callback to apply them. Use the callback to ensure that all
    ///      sibling merges are able to resolve before any of them
    ///      resolve. This keeps the parent merge pristine until such
    ///      time as it abandons or commits the merge. Pseudo
    ///      transactional behavior.
    /// </summary>
    /// <remarks>
    ///   <para>
    ///     <note type="note">***THIS IS A VALUE TYPE***</note> to
    ///     alleviate GC pressure. This will be invoked for each property
    ///     of each entity which gets merged. As such, this could see
    ///     many very small lifetimes.
    ///   </para>
    /// </remarks>
    internal struct PendingMergeResolution
    {
        /// <summary>
        ///   The commit callback to execute the resolution.
        /// </summary>
        public readonly Action Commit;

        private readonly MergeActionResult _result;

        /// <summary>
        ///   Initializes a new instance of the <see
        ///   cref="PendingMergeResolution"/> struct.
        /// </summary>
        /// <param name="result">The result.</param>
        /// <param name="commit">The commit.</param>
        public PendingMergeResolution(MergeActionResult result, Action commit)
        {
            _result = result;
            Commit = commit;
        }

        /// <summary>
        ///   Gets a value indicating whether this instance can resolve.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if this instance can resolve;
        ///   otherwise, <see langword="false"/>.
        /// </value>
        public bool CanResolve { get { return _result.HasFlag(MergeActionResult.
            Resolved); } }
    }
}
```

Listing C.24: src/Rel.Merge/PendingMergeResolution.cs

```
using System;
using System.Linq.Expressions;
using System.Reflection;

namespace Rel.Merge
{
    /// <summary>
    /// A base class for generating compiled get/set accessors for
    /// runtime resolved properties.
    /// </summary>
    public abstract class PropertyWrapper
    {
        /// <summary>
        /// Creates the specified information.
        /// </summary>
        /// <typeparam name="TClass">The type of the class.</typeparam>
        /// <typeparam name="TProp">The type of the property.</typeparam>
        /// <param name="info">The information.</param>
        /// <returns></returns>
        public static PropertyWrapper<TClass, TProp> Create<TClass, TProp>(
            PropertyInfo info)
        {
            return new PropertyWrapper<TClass, TProp>(info);
        }

        /// <summary>
        ///   Creates a property getter.
        /// </summary>
        /// <typeparam name="TProp">The type of the value.</typeparam>
        /// <param name="property">The property to wrap.</param>
        /// <returns>
        ///   A function which wraps the get method of an arbitrary
        ///   property with a native compiled function.
        /// </returns>
        /// <remarks>
        ///   This is for performance gains in runtime resolved property
        ///   accessors. This method circumvents the System.Reflection
        ///   namespace performance penalties and overhead of explicit
        ///   dynamic proxy class and assembly generation with the
        ///   System.Reflection.Emit namespace. Benchmarks for property
        ///   retrieval via native, compiled lambda expression, and via
        ///   reflection invocation measured in mm:ss.fffffff and given below.
        ///
        ///   Native: e.Prop
        ///   Lambda: e =&gt; e.prop
        ///   Reflection: propGetter.Invoke(e, null)
        ///
        ///  Iterations |     Native    |     Lambda    |  Reflection
        /// ------------+---------------+---------------+--------------
        ///       1000  | 00:00.0005833 | 00:00.0000617 | 00:00.0003298
        ///      10000  | 00:00.0004251 | 00:00.0006168 | 00:00.0031853
        ///     100000  | 00:00.0083098 | 00:00.0056037 | 00:00.0442952
        ///    1000000  | 00:00.0355637 | 00:00.0536704 | 00:00.3333400
        ///   10000000  | 00:00.3717554 | 00:00.5645168 | 00:03.0993979
        ///  100000000  | 00:03.6002561 | 00:05.3618157 | 00:31.0219926
        /// 1000000000  | 00:35.2922689 | 00:53.7391055 | 05:10.4827815
        /// </remarks>
        public static Func<TClass, TProp> CreatePropertyGetter<TClass, TProp>(
            PropertyInfo property)
        {
            var p = Expression.Parameter(typeof(TClass));

            var expr = Expression.Lambda<Func<TClass, TProp>>(
                Expression.Call(
                p,
                property.GetGetMethod()),
                p);

            return expr.Compile();
        }

        /// <summary>
        ///   Creates a property setter.
        /// </summary>
        /// <typeparam name="TProp">The type of the value.</typeparam>
        /// <param name="ccProperty">The cc property.</param>
        /// <returns></returns>
        public static Action<TClass, TProp> CreatePropertySetter<TClass, TProp>(
            PropertyInfo property)
        {
            var pe = Expression.Parameter(typeof(TClass));
            var pv = Expression.Parameter(typeof(TProp));

            var expr = Expression.Lambda<Action<TClass, TProp>>(
                Expression.Call(
                pe,
                property.GetSetMethod(),
                new[] { pv }),
                pe, pv);

            return expr.Compile();
        }
    }

    /// <summary>
    ///   A typed implementation of the property wrapper to avoid
    ///   repeated un-/boxing operations for value types.
    /// </summary>
    /// <typeparam name="TClass">The type of the class.</typeparam>
    /// <typeparam name="TProp">The type of the property.</typeparam>
    public class PropertyWrapper<TClass, TProp>
        : PropertyWrapper
    {
        private readonly Func<TClass, TProp> _getter;
        private readonly string _name;
        private readonly Action<TClass, TProp> _setter;

        /// <summary>
        ///   Initializes a new instance of the <see
        ///   cref="PropertyWrapper{TClass, TProp}"/> class.
        /// </summary>
        /// <param name="info">The information.</param>
        public PropertyWrapper(PropertyInfo info)
        {
            _name = info.Name;
            if (info.CanRead)
            {
                _getter = CreatePropertyGetter<TClass, TProp>(info);
            }
            if (info.CanWrite)
                _setter = CreatePropertySetter<TClass, TProp>(info);
        }

        /// <summary>
        ///   Gets the value stored at the wrapped property from the
        ///   specified object.
        /// </summary>
        /// <param name="obj">The object.</param>
        /// <returns>The property value of obj.</returns>
        public TProp Get(TClass obj)
        {
            return _getter(obj);
        }

        /// <summary>
        ///   Sets the given value to the wrapped property on the
        ///   specified object.
        /// </summary>
        /// <param name="obj">The object.</param>
        /// <param name="value">The value.</param>
        /// <returns>
        ///   <paramref name="obj"/> for convenience in chaining if so desired.
        /// </returns>
        /// <example>
        ///   <c>Assert.AreEqual(100, propertyWrapper.Set(ent, 100).Property);</c>
        /// </example>
        public TClass Set(TClass obj, TProp value)
```

```
        {
            _setter(obj, value);
            return obj;
        }
    }
}
```

Listing C.25: src/Rel.Merge/PropertyWrapper.cs

```
using System;

namespace Rel.Merge.Strategies
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
    public sealed class DirtyDelete : MergeableAttribute
    {
        protected internal override void Merge<TValue>(MergeAction<TValue> request)
        {
            if (request.Kind == MergeKind.DirtyDelete)
            {
                if (object.Equals(default(TValue), request.CFIM))
                {
                    //System.Diagnostics.Trace.TraceInformation("Dirty NOOP Delete");
                    request.Resolve(MergeActionResult.Resolved, default(TValue));
                }
                else
                {
                    //System.Diagnostics.Trace.TraceInformation("Dirty Delete -- >
                        Approved");
                    request.Resolve(MergeActionResult.Delete, request.CFIM);
                }
            }
        }
    }
}
```

Listing C.26: src/Rel.Merge/Strategies/DirtyDelete.cs

```
using System;

namespace Rel.Merge.Strategies
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
    public sealed class HiddenDelete : MergeableAttribute
    {
        protected internal override void Merge<TValue>(MergeAction<TValue> request)
        {
            if (request.Kind == MergeKind.HiddenDelete)
            {
                if (object.Equals(default(TValue), request.AFIM))
                {
                    //System.Diagnostics.Trace.TraceInformation("Hidden NOOP Delete");
                    // noop delete
                    request.Resolve(MergeActionResult.Resolved, default(TValue));
                }
                else
                {
                    //System.Diagnostics.Trace.TraceInformation("Hidden Delete -->
                        Create");
                    request.Resolve(MergeActionResult.Create, request.AFIM);
                }
            }
        }
    }
}
```

Listing C.27: src/Rel.Merge/Strategies/HiddenDelete.cs

```
using System;

namespace Rel.Merge.Strategies
{
    /// <summary>
    ///   Resolves conflicts by always forcing the new value on the
    ///   old value. May be applied to both classes and properties.
    /// </summary>
    [AttributeUsage(AttributeTargets.Property | AttributeTargets.Class, AllowMultiple
        = false)]
    public sealed class LastWriteWinsAttribute
        : MergeableAttribute
    {
        private readonly bool _doesLastWriteWin;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="LastWriteWinsAttribute"/> class.
        /// </summary>
        public LastWriteWinsAttribute()
            : this(true)
        {
        }

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="LastWriteWinsAttribute"/> class.
        /// </summary>
        /// <param name="doesLastWriteWin">
        ///   if set to <see langword="true"/> will operate with last
        ///   write wins logic. If <see langword="false"/> concurrent
        ///   writes are rejected. <note type="note">Defaults to <see langword="true
            "/>.</note>
        /// </param>
        public LastWriteWinsAttribute(bool doesLastWriteWin)
        {
            _doesLastWriteWin = doesLastWriteWin;
        }

        /// <summary>
        ///   Gets a value indicating whether the application of this
        ///   attribute actually results in a last write win or last
        ///   write reject.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if the last write should win;
        ///   otherwise, <see langword="false"/>.
        /// </value>
        public bool DoesLastWriteWin { get { return _doesLastWriteWin; } }

        /// <summary>
        ///   Merges the values given into the modified value.
        /// </summary>
        /// <typeparam name="TValue">The type of the value.</typeparam>
        /// <param name="baseValue">The base value.</param>
        /// <param name="current">The current.</param>
        /// <param name="modified">The modified.</param>
        /// <returns>
        ///   <see langword="true"/> if merge was successful;
        ///   otherwise, <see langword="false"/>.
        /// </returns>
        protected internal override void Merge<TValue>(MergeAction<TValue> request)
        {
            if (!_doesLastWriteWin)
                return;

            switch (request.Kind)
            {
                case MergeKind.Auto:
                    throw new NotSupportedException();
                case MergeKind.ConflictingUpdate:
                    request.Resolve(MergeActionResult.Update, request.AFIM);
                    break;

                case MergeKind.HiddenDelete:
                    if (request.AFIM != null)
                    {
                        request.Resolve(MergeActionResult.Create, request.AFIM);
                    }
                    else
                    {
                        // noop resolution
                        request.Resolve(MergeActionResult.Resolved, default(TValue));
                    }
                    break;

                case MergeKind.DirtyDelete:
                    request.Resolve(MergeActionResult.Delete, request.CFIM);
                    break;

                default:
                    throw new ArgumentException("request.Kind");
            }
        }
    }
```

```
}
```

Listing C.28: src/Rel.Merge/Strategies/LastWriteWinsAttribute.cs

```
using System;

namespace Rel.Merge.Strategies
{
    /// <summary>
    ///   The base class for class and property decoration for merging.
    /// </summary>
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Property, AllowMultiple
        = true)]
    public abstract class MergeableAttribute
        : Attribute
    {
        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="MergeableAttribute"/> class.
        /// </summary>
        protected MergeableAttribute()
        {
        }

        /// <summary>
        ///   Merges the values given into the modified value.
        /// </summary>
        /// <typeparam name="TValue">The type of the value.</typeparam>
        /// <param name="request">The request.</param>
        protected internal abstract void Merge<TValue>(MergeAction<TValue> request);
    }
}
```

Listing C.29: src/Rel.Merge/Strategies/MergeableAttribute.cs

```
using System;

namespace Rel.Merge.Strategies
{
    /// <summary>
    ///   Base implementation of merge attributes which should be
    ///   applied to numeric only field.
    /// </summary>
    [AttributeUsage(AttributeTargets.Property)]
    public abstract class NumericMergeableAttribute : MergeableAttribute
    {
        /// <summary>
        ///   Merges the values given into the modified value.
        /// </summary>
        /// <typeparam name="TValue">The type of the value.</typeparam>
        /// <param name="request">The request.</param>
        /// <exception cref="System.InvalidOperationException">
        ///   Property attributes should only be called during
        ///   conflicting updates.
        /// </exception>
        protected internal override void Merge<TValue>(MergeAction<TValue> request)
        {
            if (request.Kind != MergeKind.ConflictingUpdate)
                throw new InvalidOperationException("Property attributes should only
                    be called during conflicting updates.");
        }

        /// <summary>
        ///   Converts the value to the specified output as double.
        /// </summary>
        /// <param name="src">The input value.</param>
        /// <param name="dest">The output value.</param>
        /// <returns>
        ///   <see langword="true"/> if translation was successful;
        ///   otherwise, <see langword="false"/>.
        /// </returns>
        protected virtual bool Coherse(object src, out double dest)
        {
            dest = 0;

            if (src != null)
            {
                dest = Convert.ToDouble(src);
                return true;
            }
            return false;
        }
    }
}
```

Listing C.30: src/Rel.Merge/Strategies/NumericMergeableAttribute.cs

```
using System;

namespace Rel.Merge.Strategies
{
    /// <summary>
    ///   Allows for specifying limits to quantity and monotonicity of
    ///   change permissible during merge operations.
    /// </summary>
    [AttributeUsage(AttributeTargets.Property, AllowMultiple = true, Inherited = false
        )]
    public class StepMergeableAttribute : NumericMergeableAttribute
    {
        private readonly Func<double, double, bool> _accept;
        private readonly double _lbound, _ubound;

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="StepMergeableAttribute"/> class.
        /// </summary>
        /// <param name="isPercent">
        ///   If set to <see langword="true"/> bounding values are
        ///   taken as percentages and the change is calculated as a
        ///   percent change.
        /// </param>
        /// <param name="lbound">The lower bound of change.</param>
        /// <param name="ubound">The upper bound of change.</param>
        /// <exception cref="System.ArgumentException">
        ///   lbound must be less than ubound;lbound,ubound
        /// </exception>
        public StepMergeableAttribute(bool isPercent, double lbound, double ubound)
            : base()
        {
            if (lbound > ubound)
                throw new ArgumentException("lbound must be less than ubound", "lbound
                    ,ubound");
            _lbound = lbound;
            _ubound = ubound;

            if (isPercent)
            {
                _accept = AcceptPercent;
            }
            else
            {
                _accept = AcceptMagnitude;
            }
        }

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="StepMergeableAttribute"/> class.
        /// </summary>
        /// <param name="lbound">The lower bound of change.</param>
        /// <param name="ubound">The upper bound of change.</param>
        public StepMergeableAttribute(double lbound, double ubound)
            : this(false, lbound, ubound)
        {
        }

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="StepMergeableAttribute"/> class using
        ///   plus/minus <paramref name="step"/> for the bounds.
        /// </summary>
        /// <param name="step">The step size permissible.</param>
        public StepMergeableAttribute(double step)
            : this(false, step)
        {
        }

        /// <summary>
        ///   Initializes a new instance of the
        ///   <see cref="StepMergeableAttribute"/> class using
        ///   plus/minus <paramref name="step"/> for the bounds.
        /// </summary>
        /// <param name="isPercent">
        ///   if set to <see langword="true"/> bounding values are
        ///   taken as percentages of change and the change is
        ///   calculated by (current-afim) / (current+afim).
        /// </param>
        /// <param name="step">The step.</param>
        public StepMergeableAttribute(bool isPercent, double step)
            : this(isPercent, -Math.Abs(step), Math.Abs(step))
        {
        }

        /// <summary>
        ///   Gets or sets a value indicating whether divide by zero
        ///   should reject or resolve.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if divide by zero should resolve;
        ///   otherwise, <see langword="false"/>.
        /// </value>
        public bool DivideByZeroOk { get; set; }

        /// <summary>
        ///   Gets or sets a value indicating whether lower bound is
        ///   inclusive. Defaults to <see langword="false"/>.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if the lower bound is inclusive;
        ///   otherwise, <see langword="false"/>.
        /// </value>
        public bool InclusiveLBound { get; set; }

        /// <summary>
        ///   Gets or sets a value indicating whether upper bound is
        ///   inclusive. Defaults to <see langword="false"/>.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if upper bound is inclusive;
        ///   otherwise, <see langword="false"/>.
        /// </value>
        public bool InclusiveUBound { get; set; }

        /// <summary>
        ///   Gets a value indicating whether this instance is
        ///   percentage based.
        /// </summary>
        /// <value>
        ///   <see langword="true"/> if this instance is percentage
        ///   based; otherwise, <see langword="false"/>.
        /// </value>
        public bool IsPercentageBased { get { return _accept == AcceptPercent; } }

        /// <summary>
        ///   Gets the lower bound.
        /// </summary>
        /// <value>The lower bound.</value>
        public double LowerBound { get { return _lbound; } }

        /// <summary>
        ///   Gets the upper bound.
        /// </summary>
        /// <value>The upper bound.</value>
        public double UpperBound { get { return _ubound; } }

        /// <summary>
        ///   Merges the specified request.
        /// </summary>
        /// <typeparam name="TValue">The type of the value.</typeparam>
        /// <param name="request">The request.</param>
        protected internal override void Merge<TValue>(MergeAction<TValue> request)
        {
            base.Merge(request);

            double current, next;
            if (Coherse(request.CFIM, out current) && Coherse(request.AFIM, out next))
            {
                if (_accept(current, next))
                    request.Resolve(MergeActionResult.Update, request.AFIM);
            }
        }

        /// <summary>
        ///   Resolves if a calculated delta lies within the
        ///   acceptable bounds.
        /// </summary>
        /// <param name="delta">The delta.</param>
        /// <returns>
        ///   <see langword="true"/> if the change is acceptable;
        ///   otherwise, <see langword="false"/>.
        /// </returns>
        private bool Acceptable(double delta)
        {
            if (InclusiveLBound)
            {
                if (delta < _lbound)
                    return false;
            }
            else if (delta <= _lbound)
            {
                return false;
            }

            if (InclusiveUBound)
            {
                if (delta > _ubound)
                    return false;
            }
            else if (delta >= _ubound)
            {
                return false;
            }

            return true;
        }

        /// <summary>
```

```
        ///   The default comparison, calculates change by simple
        ///   fixed value step.
        /// </summary>
        /// <param name="current">The current.</param>
        /// <param name="next">The next.</param>
        /// <returns>
        ///   <see langword="true"/> if the change is acceptable;
        ///   otherwise, <see langword="false"/>.
        /// </returns>
        private bool AcceptMagnitude(double current, double next)
        {
            var delta = next - current;
            return Acceptable(delta);
        }

        /// <summary>
        ///   Calculates acceptable step size as a percentage.
        /// </summary>
        /// <param name="current">The current.</param>
        /// <param name="next">The next.</param>
        /// <returns>
        ///   <see langword="true"/> if the change is acceptable;
        ///   otherwise, <see langword="false"/>.
        /// </returns>
        private bool AcceptPercent(double current, double next)
        {
            if (current == 0)
            {
                return DivideByZeroOk;
            }
            var delta = (next - current) / current;

            return Acceptable(delta);
        }
    }
}
```

Listing C.31: src/Rel.Merge/Strategies/StepMergeableAttribute.cs

APPENDIX D

PROPERTY PERFORMANCE DATA

The following table provides the raw numerical data used in verifying the property access methods used in this work.

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
1	34248364	29322843	29135323	3651148
2	32140312	30878063	28610779	3595395
3	32300566	30591885	29007317	3805716
4	32337779	30122661	27540381	3681504
5	32020120	30469309	28929973	3736493
6	31762633	30308593	28747434	3771758
7	32174345	30595617	28832569	3772394
8	32075885	30696931	28881545	3772630
9	32077316	30519845	28868426	3795095
10	31973292	30777653	28696907	3790224
11	32250895	30779251	28863845	3806090
12	32150187	30736694	28975859	3773788
13	32071909	30585035	28970017	3796548
14	32214880	30362781	28731092	3792589
15	32293029	30594297	28867667	3790078
16	32100233	30810483	28806222	3818518
17	32151880	30610494	28934931	3767322
18	32215029	30484403	27646056	3688681
19	31941271	29002126	28866475	3751999

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
20	31641835	29971915	27334461	3403291
21	31312607	29538794	28085864	3763753
22	31357520	29763654	28881657	3683866
23	31355147	29422027	27962489	3759314
24	31703882	30590500	28755043	3710100
25	31248165	30159181	28410032	3648605
26	31985518	29981577	28531734	3745732
27	31670516	30730462	28706547	3485420
28	32368344	30662450	28783151	3760740
29	32115881	30715018	28936273	3680014
30	31358448	29816316	28793102	3563808
31	31841806	30381806	28912034	3767336
32	32129112	30592873	28799232	3784135
33	31939211	30646972	28721014	3762159
34	31717749	30601480	28861178	3778075
35	30930952	30857160	28936761	3724347
36	32146614	30744447	29090663	3758130
37	31727136	30591575	28953477	3633836
38	31227270	29913645	28696174	3783242
39	32082703	30508504	28397067	3707521
40	31962167	30646170	28761916	3756413
41	31656425	30592887	29057630	3805737
42	32074861	30606310	28790322	3713426
43	32034522	30051711	29255992	3725382
44	32241693	30163582	28474544	3712843
45	31708683	30328125	28669786	3712277
46	31919660	30716869	28530667	3654422

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
47	32230913	30327742	27032297	3723980
48	31857719	30474847	28783666	3610648
49	32130150	30401938	28851678	3776649
50	32010734	30364536	28744632	3644297
51	32160102	30609012	28958157	3763713
52	32337661	30585849	28682312	3797188
53	31710156	30617384	28841942	3723163
54	32135400	30570365	28987948	3816078
55	32029552	30583794	28757752	3768923
56	31043001	29583115	28801616	3740301
57	32385294	30669137	29012577	3682591
58	32201398	30672467	28737405	3754530
59	31939931	30246852	28518882	3764955
60	31075578	30067023	28758947	3747593
61	31882118	30170853	28752370	3803603
62	32337765	30507421	28718397	3694488
63	32088277	30445497	28833157	3759247
64	31776800	30111073	29015359	3700218
65	31547086	26045545	27450589	3386234
66	32150561	30639662	28934366	3738657
67	32014090	29674515	28727769	3684338
68	32222285	30568929	28765646	3777799
69	31960710	30720115	28231199	3754717
70	32118914	30645194	28927289	3775250
71	32052093	30455055	28366278	3670187
72	32028754	30381342	28736406	3739219
73	32080170	30414534	28997485	3743019

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
74	32251100	30875251	28661211	3736564
75	32606458	29894376	28848236	3650210
76	31296720	30702789	28928422	3758876
77	32384041	30184547	28763692	3774663
78	31893257	30738258	28691112	3761249
79	31678972	30578439	28744523	3757144
80	32163448	30402463	28814966	3753329
81	31996417	30578466	28750279	3732047
82	31749829	30556157	28787211	3712150
83	31382988	30076643	29023892	3686512
84	32103785	30641026	28672741	3755344
85	32265857	30575706	28574813	3734604
86	32224801	30617321	28756522	3763731
87	31798854	30365200	28601750	3699777
88	31416402	30600393	28653410	3756060
89	31846789	29716759	28553526	3479147
90	31736885	30380576	28488891	3657749
91	30910615	30540914	29016156	3585505
92	31789570	30041385	28762857	3653584
93	32102223	30489295	28443438	3694328
94	31542083	30044828	28933055	3731369
95	31365325	29930333	28613986	3687444
96	31240547	30247121	28664864	3418785
97	29385843	28101364	29002197	3589143
98	32031701	29934020	28725870	3727599
99	32383691	30402510	28717530	3756537
100	32179272	30561472	28804283	3777420

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
101	31975705	30245411	28513645	3731994
102	31765195	30589100	28768062	3758891
103	31817120	30719613	28569530	3618730
104	31926557	29846911	28815453	3628822
105	35016817	29456298	28965750	3665117
106	34697271	29120938	28902399	3763724
107	35036252	29281641	28348589	3781475
108	34885104	28901939	28721613	3723346
109	34910696	29073898	28506755	3751321
110	34339603	29336238	28736830	3698453
111	34890989	28066393	28612870	3717981
112	34944887	29534889	28948716	3606073
113	34711851	28963852	28730345	3716113
114	34963662	29712534	29143724	3762840
115	34787226	29267897	28750417	3708019
116	34672157	29213802	28453991	3652645
117	34839717	29319734	28919904	3728504
118	34969106	29377729	28663094	3732388
119	33919870	29353013	27401741	3727983
120	35031266	29688643	28499178	3726385
121	34608477	29416533	28665323	3715482
122	35037642	29407178	28328725	3643465
123	34955444	29520760	28703520	3711419
124	34549997	27773846	29139128	3572481
125	34034750	29541016	29018913	3739422
126	34727721	29489647	28501697	3779297
127	34950112	29503154	28834317	3719962

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
128	34732852	29571549	29066674	3754117
129	34838232	29635946	29034613	3715007
130	35036436	29513468	29337202	3760357
131	35060061	29494326	29353967	3786393
132	34982297	29525796	29347057	3667015
133	34888632	29553910	29330860	3687348
134	34300139	29276278	29343506	3763647
135	34916788	29166627	26330641	3733167
136	33526609	26806369	27944570	3674609
137	34509408	29399399	29102239	3776880
138	35138935	29422083	29197690	3772469
139	34616392	29627443	29338322	3705316
140	34670790	29313765	29109438	3725174
141	34875110	29604251	28861223	3745879
142	34344316	29574886	29214994	3760798
143	35039053	28837845	29396291	3774877
144	34764038	28904883	28520838	3737442
145	35149404	29622069	28844903	3783582
146	34739835	29360150	29274087	3680525
147	34058213	29515710	29318685	3757783
148	34784748	29013134	29183695	3733687
149	35200278	29019326	27789752	3260702
150	34631212	29586594	28949805	3786317
151	34030259	28641501	29653788	3712465
152	34952120	29455567	29478669	3753133
153	34862069	29107412	27637060	3658086
154	34750081	29428056	28974955	3596634

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
155	34319791	29407568	29101681	3763587
156	32065458	28436603	29200756	3765463
157	33443490	29449304	28556937	3677416
158	34428418	28353354	29173763	3595379
159	33082453	29376843	28940787	3621928
160	35095030	29477959	28823316	3630120
161	34195025	28920257	29095267	3675944
162	35048741	29506135	29470566	3718019
163	34562997	29759988	28235736	3728482
164	34983562	28810934	29212413	3785632
165	34955047	29561584	29343449	3695453
166	34134913	29517967	29200432	3779895
167	34749363	29603567	29191310	3758245
168	34530075	28955189	28939380	3689132
169	34721675	29308510	28951463	3684825
170	34847447	29532044	29024604	3725736
171	34162796	28544229	28681920	3554576
172	32246062	27652022	28011289	3757063
173	34874714	29566453	29258464	3776417
174	34910837	29644154	29178287	3767569
175	35064235	29211563	29393668	3763859
176	34984096	29234146	29034357	3720979
177	34602172	29531048	28763659	3714984
178	32170948	26899629	29023131	3719575
179	34610489	29450852	29299466	3730872
180	34600599	29280297	29394964	3687374
181	34990489	29529547	29376880	3743665

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
182	34737349	29379200	29151666	3777352
183	35055582	29649472	29323262	3783034
184	34173142	29374749	29451264	3722713
185	34622382	29558981	29606313	3709803
186	34818113	29815278	29293791	3706575
187	34239983	28772255	28494433	3797059
188	34456104	29138333	28836253	3704693
189	34881594	29476585	29234504	3742848
190	35182213	29562130	29166553	3700284
191	35221644	29667315	29412705	3792980
192	34967238	29384752	28931096	3721938
193	35124185	29555213	29271620	3747680
194	34892091	29215432	28795020	3737364
195	34802245	29785568	29602759	3723123
196	34855061	29678215	29112059	3766108
197	35062302	29626543	29124356	3758764
198	34423191	29605042	29388195	3662392
199	34539088	29318952	29135212	3748318
200	34971860	29539881	28931226	3720339
201	34718150	29676593	29373774	3686929
202	33432681	29500662	29377861	3786045
203	35065887	29379583	29238706	3733736
204	34539763	29537698	29327434	3718988
205	34763903	29382807	29383690	3753131
206	35205001	29680813	29336827	3765992
207	34934514	29455058	29031557	3665583
208	33817034	29395241	27718217	3761189

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
209	34666361	29461833	29373573	3737639
210	34616976	29547263	29346886	3773101
211	35128959	29291239	29298791	3782489
212	34896503	29422077	29102514	3727877
213	34913072	29615044	29434569	3735271
214	34996195	28686594	28674925	3566818
215	34701307	29709771	29176576	3758394
216	35349719	29606302	29073015	3700377
217	34274379	29252274	27932199	3724129
218	34879191	29611571	29471237	3536916
219	34858947	29552459	29021879	3778908
220	34759486	29610866	29259684	3737426
221	34680757	29104561	29056072	3766443
222	34800897	29728861	29052432	3743326
223	35184963	29609446	29476109	3792914
224	34709337	29729848	29442403	3782999
225	34856128	29693065	29250738	3691779
226	34588950	29122126	28541193	3621231
227	34943772	29192827	29522371	3736915
228	35108401	28461929	29243289	3709648
229	34370904	29544627	28986986	3673382
230	33822904	29305041	29251573	3743558
231	35020039	29323654	29367090	3731578
232	34227012	29508914	29080524	3451123
233	33504448	27338445	27584530	3627069
234	34917946	29100030	28921562	3665776
235	34329530	29317810	29258073	3780458

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
236	34218505	29211998	27569305	3791205
237	35100434	29771057	29427714	3731151
238	34901065	29641085	29349464	3724621
239	34186941	29522450	29471824	3703678
240	34189958	29430930	29463613	3803712
241	35104897	29562551	29138751	3752512
242	35023538	29257467	27037880	3430269
243	32725455	27928626	27953752	3721874
244	34892618	28974191	28042725	3545045
245	33246246	27468007	28695575	3668156
246	33495538	29344318	28485570	3600303
247	34955234	28970077	29073175	3678227
248	34616766	29646545	29287514	3741801
249	35044929	29307866	28860450	3758202
250	34838394	28837860	28591559	3798415
251	35246751	29453384	29264913	3730581
252	34594830	28247641	29321324	3765078
253	34729031	29375529	29215211	3706324
254	34752324	29696957	28999834	3794743
255	35093290	29570500	29019839	3765748
256	34237966	29348475	29277084	3734072
257	34889624	29511784	29358272	3752372
258	34649683	29555631	29284633	3738099
259	35045667	29465209	28149744	3773944
260	34455183	29445184	29076636	3748599
261	34863812	29565459	29460013	3762362
262	34955725	29705714	28557877	3772175

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
263	34737946	29343858	29344662	3636614
264	34064907	29547490	29254608	3717503
265	34303846	29467332	29176402	3749593
266	35067429	29304418	29350943	3782110
267	34790045	29361348	29330734	3656687
268	34588643	29566043	29274037	3713204
269	34941611	28322678	29311575	3760546
270	34793450	29422285	29344782	3761609
271	35199896	29068130	29138902	3669323
272	34876353	29457101	28365180	3651017
273	33708985	29394270	29500086	3761092
274	34812512	29373619	29532837	3765530
275	34765932	29725736	29117548	3727059
276	34107367	29594762	28458592	3702210
277	35215571	29619548	29612261	3781087
278	34819254	29122376	29179809	3710214
279	34245756	29423671	29140323	3751203
280	34839563	29363793	29447128	3758722
281	34770528	29383861	29083159	3720760
282	34682949	29502208	29059741	3757047
283	34920070	29249180	29200167	3740683
284	34820055	29048696	29265357	3722859
285	34843915	29305083	29301302	3649287
286	34350624	29201515	29362485	3769892
287	34974436	29474229	29060689	3720868
288	34700307	29276873	29356322	3592498
289	34490652	29597054	29264356	3664700

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
290	34561263	29570740	29129823	3703812
291	34822856	29506531	29414599	3794981
292	35085887	29667254	29282335	3764548
293	34961225	29522901	28479542	3694275
294	33186560	26599012	28587751	3691491
295	34792787	29215084	29188826	3789525
296	34791445	29616465	29254432	3699125
297	34902255	29375475	29135883	3698206
298	34550883	29541868	29216903	3721810
299	34611033	29508340	29439286	3767939
300	34729939	29414542	29348335	3761112
301	34746328	29478806	29148463	3739155
302	34930973	28719978	29237868	3761247
303	35123248	29725894	28937464	3609487
304	33876003	29337411	29341823	3750929
305	32966237	29209612	28697862	3704343
306	34525284	29359793	28414889	3294124
307	31498740	29589981	29136889	3732390
308	34713242	29576661	29096422	3745039
309	34831041	29554320	28824685	3673740
310	34723548	29622312	29283457	3734216
311	34857917	28336990	28153061	3729152
312	34908426	29234617	28960350	3718536
313	34606851	27574162	26434072	3370459
314	34554413	29516515	28974463	3664840
315	34807671	28905803	29185798	3745823
316	35140738	29475813	29205855	3765591

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
317	35163439	29690912	29336186	3750706
318	34875585	29421647	29522101	3761350
319	34826878	29509204	29255970	3761501
320	35025335	29498565	29329698	3761828
321	34975667	29370062	29198380	3778557
322	35027737	29479299	29285650	3789626
323	34884640	29636578	29206224	3754379
324	34373416	29784138	29101995	3766708
325	34850452	29326665	29311752	3391170
326	32782698	25860935	29080696	3654292
327	34979048	29473981	28353524	3514628
328	33270477	28298553	27848225	3668815
329	34281212	29074359	28746573	3651276
330	34621843	29210154	28667402	3679431
331	34822300	28831642	28943954	3750785
332	34858920	29759910	29309203	3729054
333	34968559	29584727	29002283	3718967
334	33545264	29448103	29645967	3778633
335	35028445	29661533	28749279	3802447
336	34859659	29271098	27290578	2979638
337	33637278	28964049	27719374	3755524
338	35031058	29498607	28710305	3596540
339	34727492	29060525	29483423	3789970
340	35010504	29452145	28854874	3769614
341	34605450	29558611	27912895	3779005
342	34695114	29075649	28952552	3692002
343	34856970	29109804	29516956	3757496

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
344	34838271	29638590	29530273	3627695
345	34186730	29710860	29227432	3740662
346	34527237	29334953	29005496	3769509
347	34937345	29626574	29092242	3773316
348	34310641	29397807	29055767	3579632
349	34427550	28001129	29207648	3758475
350	35008889	29699940	28908127	3720072
351	34753462	29419045	29013505	3778235
352	34877261	28938204	29143578	3788526
353	35079819	29102965	29443736	3757816
354	34761920	29657685	29266809	3701764
355	35166093	29586391	29231972	3800106
356	34992258	29663075	29282048	3763765
357	34335316	29715342	29174052	3783362
358	35259150	29580389	29374980	3710480
359	34070889	29306723	28521076	3755926
360	34940039	29678052	29023660	3580207
361	34826738	28733964	29265266	3747099
362	34759015	29299307	29202196	3770422
363	34730930	29496594	29391797	3771661
364	34061639	29302798	29209768	3673155
365	34866568	29456619	28880956	3654922
366	33902899	29555307	28706791	3746852
367	34677160	29541343	28827819	3768954
368	34799767	29546809	28394738	3618835
369	34838885	29519143	29594684	3727277
370	34638589	27459522	25296895	3785428

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
371	33684744	29631332	28819887	3724473
372	34451647	29527397	29336565	3762985
373	34962619	29586818	29227096	3714156
374	34757673	29228678	29067272	3782178
375	34843585	29158895	29142504	3753534
376	34111794	29384645	29282197	3724808
377	35039368	29585322	29245273	3719855
378	34285851	29441340	28928299	3619047
379	35062868	29546070	28348494	3722981
380	33623752	28591672	29274429	3674796
381	34603150	29206021	28910469	3677411
382	34337423	29571480	29175330	3767734
383	34484309	29345647	29219921	3735083
384	35006166	29571340	29194709	3778905
385	34837784	29470813	29142289	3733087
386	34866378	29673517	29329994	3747552
387	34610323	29673665	29072984	3679487
388	34948444	29491877	29447664	3784426
389	34844277	29574511	29204534	3760876
390	35085434	29537932	28904243	3710342
391	33503234	27355173	28990767	3745140
392	34468370	29396558	28484025	3548461
393	32587883	25957893	28001005	3793335
394	34209133	29698654	28853040	3802403
395	34979169	29620072	29445927	3687499
396	35109621	29628401	28983477	3707986
397	34264746	29303464	29039829	3693856

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
398	33733853	29210435	28980356	3721941
399	35039217	28470827	28210744	3757655
400	34663202	29479864	29387151	3788158
401	35110090	29573446	28158497	3754808
402	34731092	29282981	29220246	3788346
403	34202956	29565975	29068150	3791509
404	34866645	29332805	28597320	3724038
405	35020359	29573336	29276680	3729133
406	35022987	27770756	28859460	3745569
407	34980017	29365284	29451168	3805397
408	34526742	29495196	29402909	3734270
409	34553747	29255066	29203847	3715641
410	34677747	28890075	29260926	3761316
411	33593037	28810618	29660431	3761275
412	34701512	29333046	28499966	3750073
413	34908517	29324420	29172195	3773613
414	34585300	29029198	28901185	3729385
415	34817369	29619084	29086647	3646095
416	34743090	29427628	28259752	3759201
417	35049676	29491266	29385331	3779359
418	34831336	28795982	29391090	3776649
419	34828193	29020201	28552516	3699468
420	35024429	29606843	29441541	3720446
421	34645176	28881622	29248521	3780709
422	34518216	29468253	27959194	3769707
423	34135617	29650690	29312757	3758008
424	34845826	29629863	29250450	3644808

Table D.1: Property Performance Data

	Native	Derived	Lambda	Reflection
425	34437592	29063805	29116349	3760654
426	34679727	29238486	29374481	3762498
427	34698094	29544820	28741718	3741718
428	34970296	29557086	29073765	3665647
429	34823251	29494567	29344046	3721099
430	34804439	29511676	29469885	3682060
431	34627675	29702121	29323914	3697087
432	34288531	29548540	29320997	3763787
433	35179569	29356433	28898897	3747642
434	34204532	29294559	29277262	3747598
435	33905753	29474137	29129677	3755089
436	34907685	29430841	29263746	3716180
437	34834619	29699310	28966636	3681642
438	34909956	29610121	29542556	3758237
439	34874694	29538933	29306396	3628822
440	33510817	29401168	29407782	3764833
441	34964519	29428555	29445005	3752160
442	35060232	29720999	27994742	3692487
443	34869027	29580896	29199275	3733280
444	34310106	29532787	29235597	3768399
445	34427847	29564930	29336799	3678701
446	34676854	29455070	29074437	3762190
447	34338894	29509326	29630468	3734216
448	34864210	29379108	29213457	3692399
449	34900640	29502393	29232316	3773714
450	34606655	29271907	29044405	3696884

REFERENCES

[1] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Coordination avoidance in database systems. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 185–196.

[2] BARBARÁ, D., AND IMIELIŃSKI, T. Sleepers and workaholics: Caching strategies in mobile environments. *SIGMOD Rec. 23*, 2 (May 1994), 1–12.

[3] CHAN, B., SI, A., AND LEONG, H. A framework for cache management for mobile databases: Design and evaluation. *Distributed and Parallel Databases 10*, 1 (2001), 23–57.

[4] ELSHARIEF, D., IBRAHIM, H., MAMAT, A., AND OTHMAN, M. A survey of methods for maintaining mobile cache consistency. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia* (New York, NY, USA, 2009), MoMM '09, ACM, pp. 351–355.

[5] FANG, Y., AND LIN, Y.-B. Strongly consistent access algorithms for wireless data networks. *Wireless Networks 11*, 3 (2005), 243–254.

[6] FAWAZ, K., AND ARTAIL, H. Dcim: Distributed cache invalidation method for maintaining cache consistency in wireless mobile networks. *IEEE Transactions on Mobile Computing 12*, 4 (Apr. 2013), 680–693.

[7] FOWLER, M., RICE, D., FOEMMEL, M., HIEATT, E., MEE, R., AND STAFFORD, R. *Patterns of Enterprise Application Architecture*. Pearson Education, Inc., 2003, ch. Offline Concurrency Patterns, pp. 415–453.

[8] HÄRDER, T. Observations on optimistic concurrency control schemes. *Inf. Syst. 9*, 2 (Nov. 1984), 111–120.

[9] JOY, P., AND JACOB, K. A key based cache replacement policy for cooperative caching in mobile ad hoc networks. In *Advance Computing Conference (IACC), 2013 IEEE 3rd International* (Feb 2013), pp. 383–387.

[10] LEE, G., JANG, I., AND PACK, S. Fast wireless data access scheme in wireless networks. In *Computing, Networking and Communications (ICNC), 2013 International Conference on* (Jan 2013), pp. 40–44.

[11] LEE, S., HWANG, C.-S., AND YU, H. Supporting transactional cache consistency in mobile database systems. In *Proceedings of the 1st ACM International Workshop on Data Engineering for Wireless and Mobile Access* (New York, NY, USA, 1999), MobiDe '99, ACM, pp. 6–13.

[12] MADHUKAR, A., ÖZYER, T., AND ALHAJJ, R. Dynamic cache invalidation scheme for wireless mobile environments. *Wirel. Netw. 15*, 6 (Aug. 2009), 727–740.

[13] QIAN, F., QUAH, K. S., HUANG, J., ERMAN, J., GERBER, A., MAO, Z., SEN, S., AND SPATSCHECK, O. Web caching on smartphones: Ideal vs. reality. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 127–140.

[14] SIVARAMAN, A. Diploma: Consistent and coherent shared memory over mobile phones. In *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012)* (Washington, DC, USA, 2012), ICCD '12, IEEE Computer Society, pp. 371–378.

[15] VOGELS, W. Eventually consistent. *Queue 6*, 6 (Oct. 2008), 14–19.

[16] WANG, X., AND FAN, P. A strongly consistent cached data access algorithm for wireless data networks. *Wireless Networks 15*, 8 (2009), 1013–1028.

[17] WANG, Z., DAS, S., CHE, H., AND KUMAR, M. A scalable asynchronous cache consistency scheme (saccs) for mobile environments. *Parallel and Distributed Systems, IEEE Transactions on 15*, 11 (Nov 2004), 983–995.

[18] XU, W., WU, W., WU, H., CAO, J., AND LIN, X. Cacc: A cooperative approachto cache consistency in wmns. *Computers, IEEE Transactions on 63*, 4 (April 2014), 860–873.

VITA

Ryan Linneman was born on October 12, 1982, in Columbia, Missouri, USA. He was home schooled until secondary school and graduated in 2001. Working full time as a software developer, he graduated in 2008 with a Bachelor's degree in Computer Science. Upon completion of this degree he enrolled at the University of Missouri - Kansas City, where he completed a second Bachelors in Computer Science in 2011 and promptly rolled into the Master's program there.

Starting from a young age Ryan took a strong interest in technology and software. He entered the workforce straight out of high school and began work as a software developer in 2004 with Burns & McDonnell Engineering driving development on their document management and CAD modeling systems. In 2013 he took up his current post with Trabon Solutions, LLC. continuing to pursue his passions in technology. While there, he has developed data integration systems, E-commerce websites, CMS/CRM portals, and a variety of other LOB products on Android, Linux, and Windows.