RC25523 (WAT1502-038) February 12, 2015
Computer Science

IBM Research Report

Integrator: An Architecture for an Integrated Cloud/On-Premise Data-Service

Avraham Leff, James T. Rayfield
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Research Division
Almaden – Austin – Beijing – Brazil – Cambridge – Dublin – Haifa – India – Kenya – Melbourne – T.J. Watson – Tokyo – Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Many reports are available at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Integrator: An Architecture for an Integrated Cloud/On-Premise Data-Service

Avraham Leff, James T. Rayfield
IBM T.J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights, NY 10598
Email: {avraham, jtray}@us.ibm.com

***Abstract*—Large enterprises have built very large "on-premise" data-sets that are critical to many business functions. With the availability of cloud-based storage, many of these enterprises are considering whether and how to make some of this data available on the cloud. One motivation is to offload the processing of new mobile application workloads from the on-premise system to the cloud. Another motivation is to improve the performance of these mobile applications. However, because of the importance of this data, and because of regulatory constraints, many enterprises are unwilling to simply move their data from an on-premise environment to the cloud. Instead, they prefer to keep the "master" version of the data on-premise, while projecting a subset of the data to the cloud.**

Several challenges face these enterprises. First, how can large data-sets be efficiently made available on the cloud with minimal disruption to the ongoing on-premise business function? Second, how can this data be represented in a way that will be useful to cloud developers? Typically, cloud developers want data represented in a way that is easily consumable by REST APIs, but the on-premise representation may not be amenable to such usage. Our INTEGRATOR project addresses these challenges by providing an integrated cloud/on-premise data-service. Importantly, the INTEGRATOR architecture is broadly applicable across various back-end systems. In this paper we describe the INTEGRATOR architecture and a prototype implementation for a specific on-premise system. We examine alternative architectures – "table based" and "business object based" – and explain why we chose the business object approach.

I. INTRODUCTION

Over time, large enterprises build *data warehouses* that store data-sets containing tens of millions of records [1]. These data-sets provide critical business function and are accessed through programs that represent the core business logic for that enterprise. Recently, with the availability of cloud-based storage and servers, many of these enterprises are considering moving or copying these on-premise data-sets to the cloud. Typically, this migration involves two types of cloud services: using a Database-as-a-Service (DBaaS) (e.g., [2], [3]); and using a mid-tier REST service (e.g., Node.js® [4] hosted on Bluemix™ [5] which accesses the DBaaS as part of the API implementation. Motivations include seeking to reduce the cost of business operations; offloading new mobile application workloads from the on-premise system to the cloud; and improving the performance (e.g., latency, throughput, and availability) of new mobile applications. Such migrations are becoming more plausible with the creation of multi-tenant, large-scale DBaaS systems [6].

At this time, however, many of these enterprises are cautious about actually *moving* these important data assets from on-premise to the cloud. For example, security considerations or regulatory requirements may require that the master or primary copy of the data continue to reside on-premise. Or, the data warehouse software may not be able to run correctly on a given cloud software stack. Instead, enterprises may prefer to only *cache* subsets of their on-premise data in the cloud. With this decision, the enterprise must decide the best strategy for loading the on-premise data to the cloud, and keeping the cloud cache synchronized with the on-premise master copy in a way that:

- minimally disrupts the ongoing operation of the on-premise systems, and
- makes the cloud data available in a useful form for its consumers, typically mobile application developers.

The second requirement is especially important as enterprises evolve from "systems of record" to "systems of engagement" [7], because it raises the issue that the on-premise data representation may not be in the right form for the cloud cache.

Our work therefore should be viewed as investigating a variant of *hybrid* cloud/on-premise architectures (e.g., [8], [9]) where the enterprise constrains the cloud portion of the hybrid architecture to be only a read-only cache of on-premise data. We have examined these issues in our INTEGRATOR prototype and implemented a solution within a specific set of technology capabilities and constraints. In this paper, we explain why we believe that the INTEGRATOR approach is broadly applicable to a range of data-warehouses and technologies. We discuss alternative architectures that we considered, and illustrate some of the benefits of our hybrid cloud/on-premise data service by describing a mobile application that we deployed to the INTEGRATOR environment. Our paper is therefore structured as follows. Section II discusses alternative strategies for replicating on-premise data to the cloud. In Section III, we explain the details of our INTEGRATOR prototype and describe a mobile application that uses the hybrid cloud/on-premise cache. We conclude by summarizing our work and mentioning possible extensions to INTEGRATOR in Section IV.

II. ALTERNATIVE REPLICATION STRATEGIES

In this section we present and contrast alternative strategies for replicating on-premise data to the cloud. We also explain why the characteristics of the on-premise data led us to adopt the second strategy in our INTEGRATOR prototype.

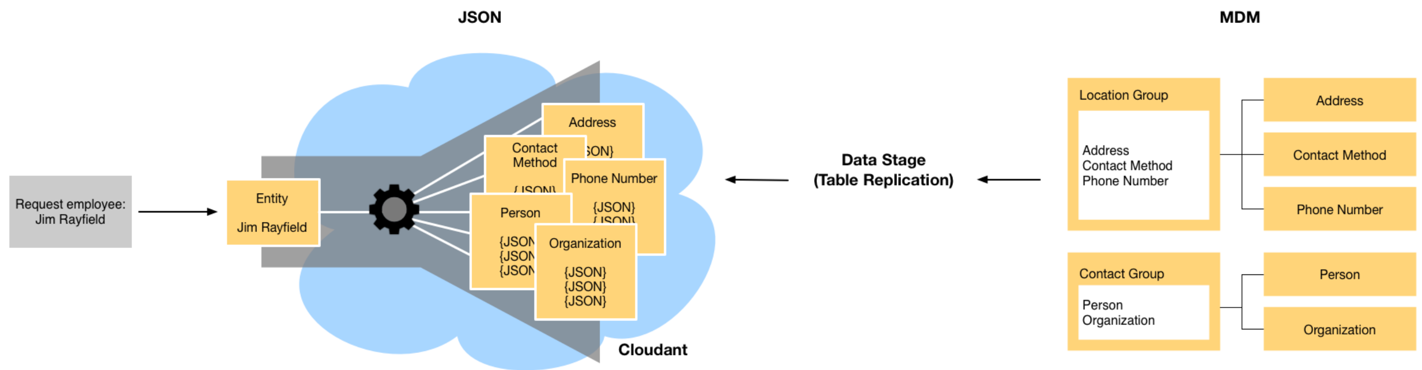

Fig. 1. "Table Based" Replication

A. *Table-Based Replication*

Our initial prototype used a *table-based* replication approach from the on-premise system to the cloud (see Figure 1). The approach is straight-forward: first, identify the set of database tables to be cached; then, copy (and keep in sync) these tables to the cloud. This approach leverages the mature set of "change data capture" (or *CDC* [10]) technologies to:

1) detect that a data-event of interest (e.g., "create", "update", "delete") has occurred
2) trigger the execution of specific code to respond to that event (e.g., by copying the data that was just created on the on-premise system to the cloud-cache).

From a performance viewpoint, the table-based replication approach has the advantage of being able to do the initial bulk copying of large data-sets efficiently, while also efficiently implementing the subsequent trickle-sync activity. In fact, this approach works well for "simple" data-sets: i.e., where little business logic is need to use the raw data in a given business function. For example, if a PERSON is represented in the on-premise system as a single tuple – regardless of the number of database columns – copying large sets of PERSONs to a cloud database works well. Whether or not the cloud database presents a relational API or a NoSQL API such as Cloudant [2], application business logic can manipulate PERSON instances using only the cloud database's representation. We found, however, that table-based replication breaks down for "complex" data-sets. By "complex" we mean that the on-premise's existing applications do not directly access or manipulate the raw data-sets. Instead, applications access the data through a substantial business logic layer that is packaged as part of the data-warehouse itself. It is precisely this type of mediating business logic layer that systems of record build into their systems, making the table-based replication approach impractical. For such systems, the data are **not** the application.

To see why, consider Figure 1. Here the on-premise system is IBM's Master Data Management (MDM [11]) product. MDM manages data about business entities such as customers, products, and accounts; enterprises use MDM to present a common view of such important business entities whose underlying data are stored and replicated across IT systems. When attempting to replicate MDM PERSON instances to the cloud, we found that the underlying data is stored in multiple database tables with complex business rules governing, for example, the relationship between a person's address and the person's phone number. When a client invokes a simple API such as "query by last name", the on-premise system must perform multiple database join operations while enforcing business rules about concepts such as address and organization cardinality, in order to assemble a PERSON entity that can be returned to the client. Thus, after doing table-based replication of PERSONs to the cloud (the middle of Figure 1) as a set of JSON documents, we found that we could not provide an API for applications to "retrieve person" without duplicating the business logic of the on-premise system. This is an error-filled, expensive, process especially in situations where the cloud-service software stack differs from the on-premise system.

B. *Business-Object Replication*

The fact that enterprises invest considerable effort in presenting a "business object" (or "entity") API on top of the underlying data suggests that a replication strategy should leverage this existing investment. After all, the on-premise system already knows how to assemble a PERSON correctly from the various database tables. Therefore, rather than replicating the raw data to the cloud, enterprises should replicate business objects such as PERSONs to the cloud. The business-object replication strategy is shown in Figure 2: MDM assembles a PERSON instance from the on-premise data, and replicates the business object to Cloudant as a JSON document. In contrast to the table-based replication approach of Figure 1, when a client requests a given PERSON instance from the mid-tier cloud server, the instance is already assembled on the cloud server and can be immediately returned by the server. From a performance perspective, this approach pays the cost of assembly "up-front" in contrast to the table-based strategy which lazily assembles the instance from the cache as necessary. The key point, however, is that the on-premise's business logic is *not* replicated to the cloud, a less costly, and less error-prone approach. The relative disadvantage of business-object replication occurs if new cloud applications wish to use the raw data in ways that are not anticipated by the on-premise API. In that case, applications will have to "disassemble" the PERSON instance, perhaps injecting new business logic to join the data with other data sources. However, in such scenarios, we anticipate that the cloud server will itself create the new APIs, removing that burden from the applications themselves. Typically, the enterprise's business needs require that the existing use cases – as reflected by the *existing* on-premise APIs– be projected to the cloud before developing

new requirements. As a result, the business-object replication strategy is a better strategy for developing a hybrid cloud/on-premise data-service

We discuss the details of the INTEGRATOR implementation in Section III. At a high-level, the on-premise system must satisfy the following requirements for business-object replication to be feasible. For every class of business object B_OBJECT_i, the on-premise system must:

1) be able to specify a list of identifiers corresponding to the business object instances that will be copied to the cache.
   Combined with the next requirement, this requirement makes it possible to do an initial "bulk-load" of the on-premise business objects to the cache.
2) have an API through which an instance with identifier id_j can be retrieved.
   The details of the API (e.g., JSON *versus* XML or HTTP *versus* RMI are of secondary importance.
3) be able to detect when a life-cycle event (e.g., "create", "update") has occurred to business object instance id_j and to "publish" both the event and the id.
   Implementation details for this requirement (e.g., "pub-sub" *versus* polling) do matter, but are less important than the basic capability on the part of the on-premise system.
   This requirement makes it possible to efficiently perform ongoing "trickle-sync" operations, through which the cloud cache is kept in sync with the on-premise system. In combination with the second requirement, the cache can update an existing business object or propagate the creation of a new instance on the on-premise system. The requirement also makes it possible for "delete" events to be propagated from the on-premise system to the cache.

III. INTEGRATOR

In this section, we present the details of our INTEGRATOR prototype, and show how rich mobile applications can be created when this sort of hybrid cloud/on-premise data-service is available. A more detailed description and pointer to sample code is available [12].

As shown in Figure 3, MDM [11] is the on-premise system whose data we integrated into the hybrid data-service. In our prototype, we focused copying a set of PERSON instances to a Cloudant [2] database. Using a straightforward SQL query, on-premise system administrators can generate a set of PARTYIDs for the set of PERSONs that are to be cached in the cloud. Recently, MDM added a set of REST APIs through which a PERSON instance with given PARTYID can be retrieved from MDM as a JSON document. These capabilities satisfy the first two requirements for business-object replication that we listed in Section II-B. In addition, MDM provides a "message queue" through which its business object implementations can publish events such as "created person" or "updated person"; clients can subscribe to this event queue using the JMS [13] API. This satisfies the third requirement of Section II-B.

The use of DataStage® [14] in Figure 3 is an example of using high-level visual programming to simplify difficult system integration tasks. We use DataStage to implement both the initial "bulk load" of PERSON instances into the cloud and the subsequent "trickle sync" phase in which the cloud cache maintains its synchronized state with the on-premise system.

DataStage consists of a visual data-flow design tool and a parallel run-time engine for design and deployment of Extract, Transform and Load (ETL) -style jobs. In the ETL jobs described next, we are using various function blocks, including: relational table iteration, REST call origination, data-flow switching, data-flow gathering, Java™ interface, computational, and sequential-file output blocks. After the DataStage job is compiled, it is loaded into the engine and executed.

A. Bulk Load

The *Bulk Load* job (Figure 4) is fairly straight-forward. The *contact_table* block reads the *contact* table in MDM. The contact table contains the PARTYIDs of both PERSONs and ORGANIZATIONs. The *party_id_type* carries both the PARTYID and party type to the *party_load* block, which is a *Hierarchical Data* stage. party_id_type also carries a constant '1' for the MDM inquiry level.

The *Hierarchical Data* stage is primarily used to compose JSON and XML, make REST calls, and parse the JSON or XML results. The REST substage is completely configurable as to the use of SSL, authentication, reuse of connections, etc. The *Hierarchical Data* stage also has various transformation stages, including *H-Pivot*, which we use to convert a multi-valued record into several XML entities.

The request is sent to MDM in XML format. The first step of party_load is an H-Pivot to place the three values on the same element. The next stage is an *XML_Composer* step named *Build_getParty*. This generates a *TCRMService* element, which is populated by the incoming values. The *TCRMInquiry* element is *getParty*.

The third stage is the actual REST call to MDM. The fourth step tests whether the REST call was successful. If the call was successful, the fifth step is a *JSON_Parser* step. The sixth step tests for *SUCCESS* in the *ResultCode* element.

The party_load block has two outputs: *json_body* and *person_org*. The first output is the returned JSON body text, and the second is either 'P' (for person) or 'O' (for organization).

The next step is a *Switch* step named *person_org_switch*. This step divides the JSON bodies between PERSONs and ORGANIZATIONs. For PERSONs, the data flow is to the *person_parse_and_insert* step, which is a *Hierarchical Data* stage. person_parse_and_insert first does a JSON parse with all the possible parameters of a getParty response when called for a PERSON party. Then it builds a Cloudant *insert* JSON string, and POSTs the string to Cloudant. Finally, the Cloudant REST status is checked, along with the HTTP status code. Any error returns are sent to the output.

The *org_parse_and_insert* step is similar, except that it parses the JSON for ORGANIZATIONs. Finally, any errors from person_parse_and_insert or organization_parse_and_insert are gathered by a *Funnel*, and written to a log file.

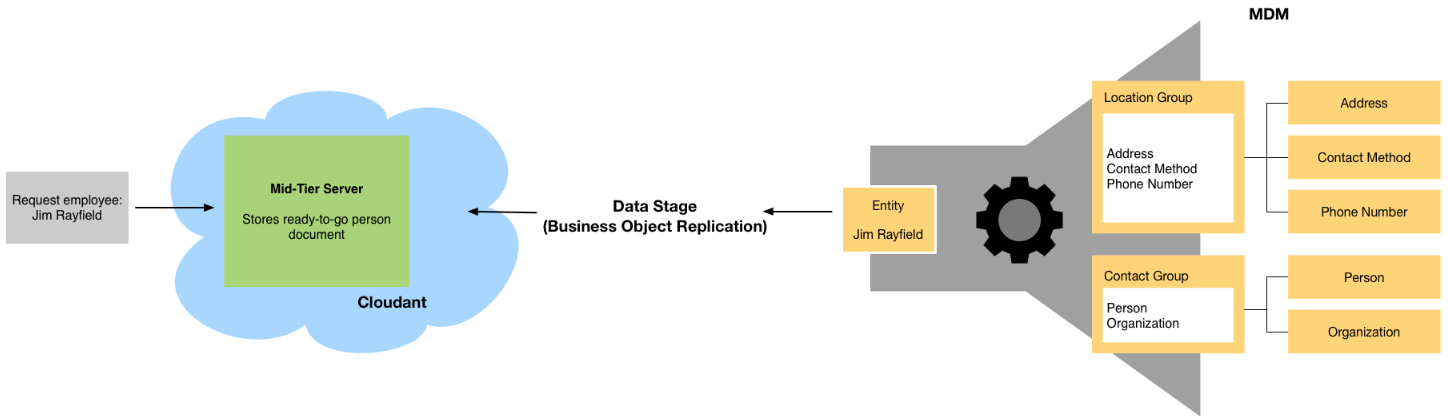

Fig. 2. "Business Object" Replication

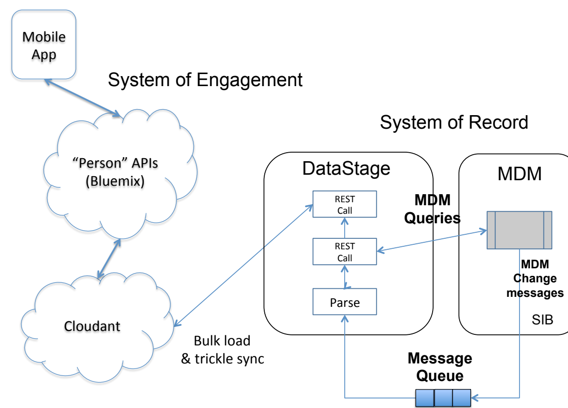

Fig. 3. Integrator Prototype Implementation

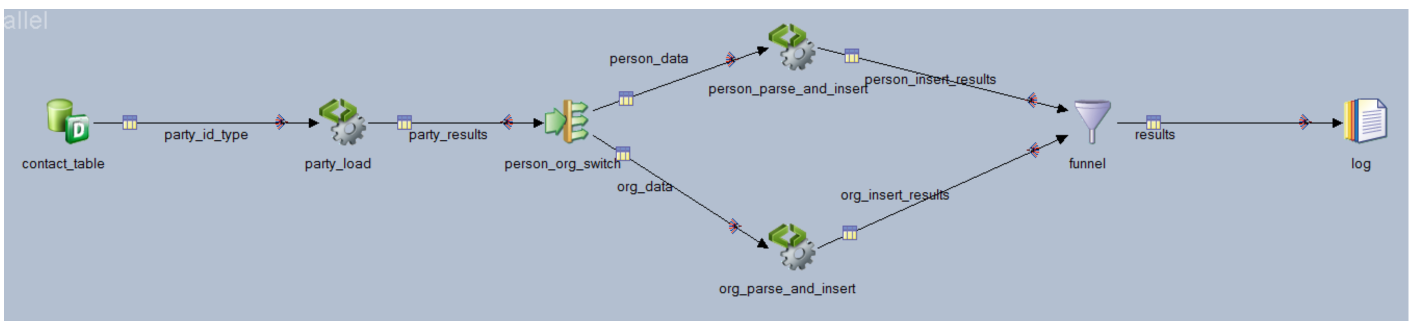

Fig. 4. Using DataStage for Initial Bulk Load of Data

B. Trickle Sync

The *party_trickle_load* job (Figure 5) is a bit more complicated. The initial stage is a Java stage which contains a *JMS Listener* that listens for MDM *change broadcast* messages. The Java stage parses each message, searching for the PARTYID, PERSONID, or ORGANIZATIONID. The Java stage passes the id and the transaction name to the next stage. The transaction name (addPerson, updatePerson, addOrganization, or updateOrganization) is extracted from the XML message.

The next stage is a *Transformer* stage called *add_PartyType_InquiryLevel*. This stage adds the party type converted to a single character ('P' or 'O') and an inquiry level fixed at '1'.

The next stage is a *Hierarchical Data* stage called *party_load*. The stage starts with an *H-Pivot* step to put PARTYID, PartyType, and InquiryLevel on the same element. Then it builds a TCRMService call with TCRMInquiry of getParty to load the item that was added or updated from MDM. The full current state of the PERSON or ORGANIZATION is loaded on each add or update message, regardless of what was changed. This makes it easier to add/update the Cloudant

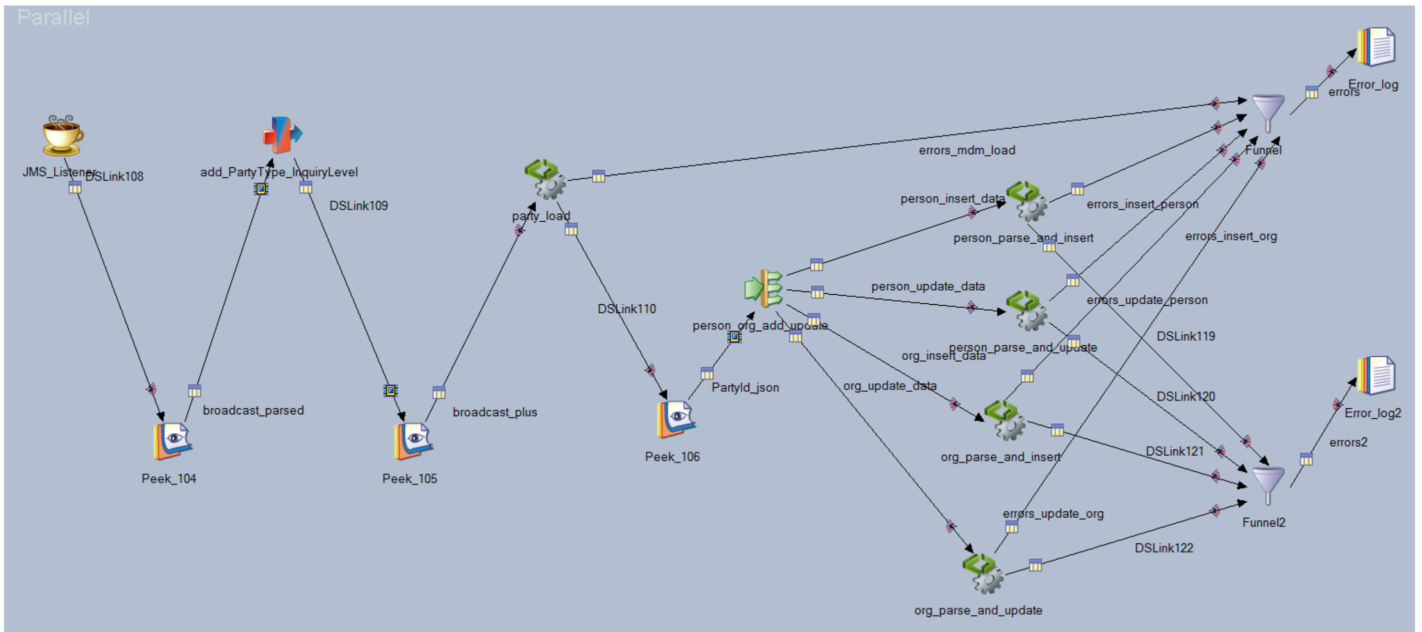

Fig. 5. Using DataStage for "Trickle Syncronization" of Data

record.

The next step is a HEAD call to MDM. This is used to reestablish the HTTP connection to MDM in case it has timed out. Then a getParty is executed against MDM, and the status of the REST call is checked.

The next stage is a *Switch* stage named *person_org_add_update*. This stage routes the data to one of four *Hierarchical Data* stages: *person_parse_and_insert*, *person_parse_and_update*, *org_parse_and_insert*, or *org_parse_and_update*. We will first consider person_parse_and_insert.

person_parse_and_insert first does a JSON parse of the body returned by MDM. If MDM returned ResultCode *SUCCESS*, it composes a Cloudant *insert* of the MDM record. Then a HEAD call is made on the Cloudant HTTP connection to recover from connection timeouts. Finally the Cloudant insert is executed, and the response is tested for success.

person_parse_and_update is more complicated. First, a JSON parse of the MDM body is done, and the ResultCode is checked. Then a HEAD call is made on the Cloudant connection in case it has timed out. Then, Cloudant is queried to find out what the current *revision* ID is for the Cloudant record. The current revision is needed for the Cloudant update call. The Cloudant query is parsed to determine the revision ID, and then the ID is mixed into the MDM data to determine the Cloudant update JSON string. Finally the Cloudant update REST call is executed, and the return code tested.

The *org_parse_and_insert* and *org_parse_and_update* are very similar. Finally, all the MDM errors are collected by a Funnel stage, and all the Cloudant errors are collected by another Funnel, and each Funnel is written to a separate log file.

C. Mobile Application

One of the benefits of an integrated cloud/on-premise data-service such as INTEGRATOR is that it allows enterprises to quickly project their data assets into the cloud so that it can be used to create new *systems of engagement* (SOE) [15], [16]. These systems of engagement provide social and collaborative features on top of existing business data from *systems of record* (SOR) to provide additional value to the consumer [7]. For example, consider an enterprise that has a purchase-order application with a critical dependency on well-defined PERSON entities. Now, the enterprise wants to create an "engaging" mobile application that will allow clients to create purchase-orders (by connecting to the existing SOR application) and – in addition – associate purchase-orders with a "chat" feature that allows clients to directly ask for help as they create the purchase-order.

Rather than enhancing the existing SOR to include a chat feature, it will typically be easier to create the system of engagement as a cloud-service using the PERSON data from the cloud data-service. With the INTEGRATOR approach, the enterprise can restrict sensitive parts of PERSON data to the existing (controlled) SOR applications, and only cache portions of the PERSON on the cloud. As shown in Figure 3, we took this approach in a demo mobile application. We created a set of REST APIs, hosted in BlueMix [5] through which mobile developers can easily consume the PERSON data; in addition, the mobile application is hosted as a Bluemix service, removing workload from the on-premise system. This approach simplifies the task of linking the chat system to the purchase orders as an SOE deployed as a cloud service, decoupled from the existing SOR.

In Code-Sample 1, we show some of the REST APIs used to construct the mobile application. Although the DataStage jobs extract a JSON document from MDM, the JSON is mapped very closely to the original, deeply nested, XML document making

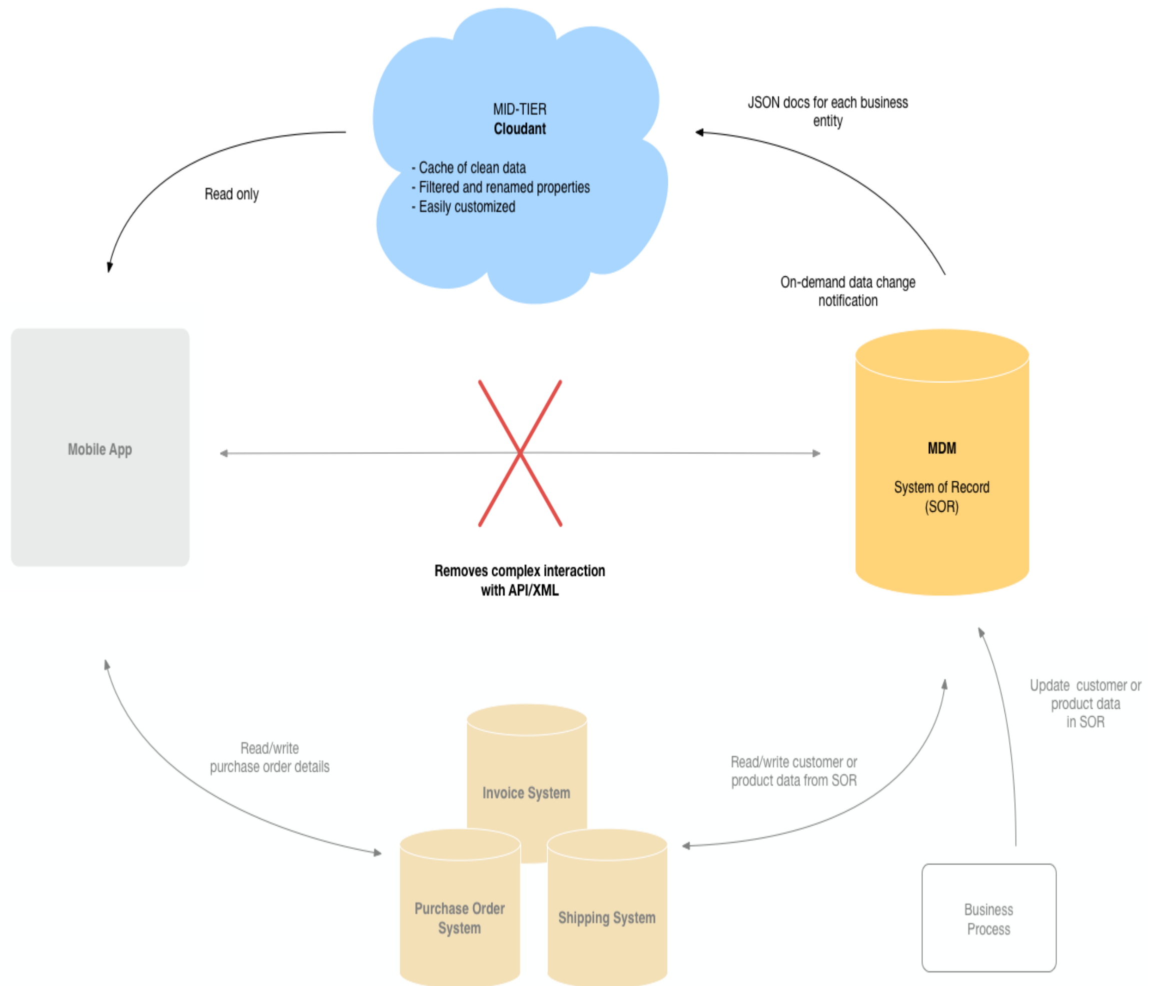

Fig. 6. Sample Integrator Mobile Application

Code Sample 1 Cloud APIs to Simplify Consumption of On-Premise Data

```
curl -i -X GET -H "Accept: application/json" http://mdmcloudantsample.mybluemix.net/mcs/person/nnn

curl -i -X GET -H "Accept: application/json" http://mdmcloudantsample.mybluemix.net/mcs/org/nnn

curl -i -X POST -H "Accept: application/json" http://mdmcloudantsample.mybluemix.net/mcs/search/person/?name=xxx

curl -i -X POST -H "Accept: application/json" http://mdmcloudantsample.mybluemix.net/mcs/match/person?last_name=xxx
```

it difficult for developers to navigate the complex document. Our REST APIs simplify this task. For example, they allow a client to retrieve a PERSON by invoking the /person API, and supplying a PARTYID parameter. A corresponding API is used to retrieve an organization instance by supplying a PARTYID. The search APIs enable a client to retrieve the set of all PERSON instances that match a given last name or the set of all PERSON instances that match a concatenation of a first and last name.

Figure 6 sketches the data-flows of a mobile application we created to use our INTEGRATOR prototype. The far-left of the Figure takes the viewpoint of a mobile developer who (1) wants to consume the valuable SOR data hosted in the on-premise MDM system but (2) would prefer not to have to understand the APIs made available by the on-premise system which produces complex XML documents. Deploying INTEGRATOR satisfies the mobile developer's requirements: interactions with the on-premise system are replaced with REST API interactions with the mid-tier that produce easily consumable JSON documents. At the same time, the business processes of the on-premise system continue to function without being affected by the new mobile applications. They continue to create and modify the MDM data using the existing business logic that provide the high-quality data required by the enterprise. On the far right, through the business-object replication algorithms described above, MDM keeps the cloud cache synchronized with its own state.

IV. Summary & Future Directions

Our INTEGRATOR prototype presents a "business-object" replication design pattern through which enterprises can efficiently construct an integrated cloud/on-premise data-service. We validated the design pattern in a concrete implementation using IBM's MDM product to cache PERSON and ORGANIZATION business entities in a Cloudant cache. Importantly, we showed that the requirements for this design pattern can typically be met by many on-premise systems because they have increasingly created web APIs that make this integration feasible.

The value of the INTEGRATOR design-pattern is that it is a low-risk approach through which sub-sets of critical enterprise data – which may be subject to regulatory or other constraints – can be safely made available for cloud consumption. New mobile applications can then be created without imposing new workload stress on the on-premise system. We also showed how cloud APIs can be constructed to make it easier to use the on-premise data, especially when integrating the system-of-record data with system-of-engagement functions.

One direction we are considering taking with the INTEGRATOR prototype is to see whether, and to what extent, concrete implementations across on-premise systems with different data-warehouse products can use a common infrastructure layer. We think this may be possible despite the fact that business-object replication depends so critically on the semantics of the business APIs.

V. Acknowledgements

The authors would like to thank: Lena Woolf, for supervising the project and giving us insights as to how MDM can best leverage the hybrid on-premise/cloud architecture; Joseph Tsang for writing the MDM REST interfaces, organizing the sample release, and numerous other tasks; and Dan Wolfson for insisting that we confront the real-world challenges of data-warehouse products.

References

[1] C. Bontempo and G. Zagelow, "The ibm data warehouse architecture," *Communications of the ACM*, vol. 41, no. 9, pp. 38–48, 1998.

[2] "Technical overview: Anatomy of the cloudant dbaas," https://cloudant.com/resources/white-papers/, 2015.

[3] C. Curino, E. P. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: A database-as-a-service for the cloud," *http://dspace.mit.edu/handle/1721.1/62241*, 2011.

[4] T. Hughes-Croucher and M. Wilson, *Node: Up and Running*. O'Reilly, 2012, iSBN:978-1-4493-9858-3.

[5] "Explore ibm bluemix," http://www.ibm.com/developerworks/cloud/library/cl-bluemix-dbarnes/, 2013.

[6] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: current state and future opportunities," in *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, 2011, pp. 530–533.

[7] J. Bersin, "The move from systems of record to systems of engagement," *Forbes*, 2012.

[8] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *Internet Computing, IEEE*, vol. 13, no. 5, pp. 14–22, 2009.

[9] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena, "Intelligent workload factoring for a hybrid cloud computing model," in *Services-I, 2009 World Conference on*. IEEE, 2009, pp. 701–708.

[10] R. K. . J. Caserta, *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, 2004.

[11] "Master data management," http://www-01.ibm.com/software/data/master-data-management/, 2015.

[12] "Using the mdm and cloudant integration sample for mobile computing," http://www-01.ibm.com/support/knowledgecenter/SSWSR9_11.4.0/com.ibm.swg.im.mdmhs.cloudant.doc/topics/mdmcloudant_intro.html, 2015.

[13] D. A. C. Mark Richards, Richard Monson-Haefel, *Java Message Service*, 2nd ed. O'Reilly Media, 2009.

[14] "Infosphere datastage," http://www-03.ibm.com/software/products/en/ibminfodata, 2015.

[15] G. Moore, "Systems of Engagement and The Future of Enterprise IT: A Sea Change in Enterprise IT," pp. 1–14, Jan. 2011.

[16] Forrester Consulting, "Systems Of Engagement Demand New Integration Solutions — And A New IT," pp. 1–20, Apr. 2013.