Created September 23, 2012, last modified November 4, 2012

Aggregation and Classification with Hadoop and HBase

By Romel Rivera,

CEO, eKnowlogie

Table of Contents

1 Introduction

CardProfit is a strategic business intelligence software application for the banking industry, used to establish and manage the profitability of credit card and payments products in multidimensional segmented markets. Segments, represented through a collection of standardized unit indicators, can then be universally compared in order to establish benchmarks and isolate potential value identification to support goal setting, goal valuation and monitoring. See www.eknowlogie.com/CardProfit.

Business performance can be isolated by simultaneously segmenting the market by credit card product, product type, cardholders' geographical location, age groups, income, payment habits, lifestyle, transactional volume, marketing campaign account harvests, time series etc. These segment intersections can then be compared very effectively. In addition, arbitrary grouping segment hierarchies can be created for each segmentation in order to find the largest segment groups where clear differentiators apply. For example, geographical location can respond to hierarchies by zip code, county, state and country, or by rural, agricultural, industrial and high-tech regions. Credit card products can respond to hierarchies by brand or by product type. Time series can respond to hierarchies by day, week, month, quarter and year, or by shopping seasons.

Because of the large volume demands, it was determined that for the new CardProfit version, operational, segmentation and intelligence information would be persisted using a distributed parallel architecture with Hadoop and HBase, likely under an elastic cloud computing framework such as Amazon AWS EMR. This would be a distributed system where multiple credit card issuers could maintain their own localized clusters and still belong to larger multi-issuer clusters where national and international standard indicators could be derived, (under certain rules for jurisdiction and anonymity).

1.1 Distributed Big Data Store Selection

Cassandra [Lakshman09], based on Dynamo [DeCandia07], offers a very elegant, truly distributed architecture based on two concepts, the ability of node peers to establish a partial ordering of events using vector clocks (which stems from Leslie Lamport's beautiful seminal paper on the subject [Lamport78]), and consistent hashing, the ability for any node in the cluster to locate data hosted at any other node. While Cassandra offers a distributed architecture with no single points of failure, it does not offer a native aggregation framework which is indispensible in order to support our analytical extractions. HBase [George11] on the other hand is designed to provide extensive native support for map reduce, a distributed framework for big data aggregation. HBase does have a JobTracker node that represents a single point of failure, but along with the NameNode server they provide flexibility to manage region splits to support "configurable sharding" which is a very desirable complement to map reduce in order to optimize the localization of mapper jobs specially when these mappers require the use of multiple input tables. This requirement to manage partitions is best served with the ordered partitioning supported by HBase. Cassandra, by contrast, favors random partitioning which would make it impossible to implement the envisioned localized multi-table mappers of many map reduce jobs. In other words, HBase is more suited for data processing and analytics (aggregation) while Cassandra more suited for real-time transaction processing. While Cassandra's eventual consistency will provide identical behavior to HBase's strong consistency for a primarily batch update system like ours, Cassandra's lack of cell versioning support would require yet additional explicit support for the representation of historical state. So the decision is made for us: HBase.

2 Problem Domain

This problem domain belongs to a very general data aggregation problem category: Domain analytics over multidimensional and multi-hierarchical data aggregation.

Aggregate data provides bird's eye views of a space or problem domain. In order to achieve differentiation in the problem space, aggregate information needs to be classified or subdivided according to one or more variables, e.g. the number of sales of a given retail product within a geographic location and purchased by consumers within a certain income bracket, education level, hobby profile and during a given time period. If we see each such variable as a dimension in a n-dimensional Cartesian space, these aggregates correspond to the Cartesian products of that space. In market analysis, if we see each variable as a market segmentation, these aggregates correspond to segment intersections.

Even though Cartesian products give us a most valuable microscopic differentiation between aggregate information, they also create a combinatorial explosion of values. Domain analytics will further benefit by characterizations made about these variables at different levels of abstraction or grouping levels so we allow any number of hierarchies to be defined over each variable, dimension or segmentation. These grouping levels may also be essential in the accurate creation of domain models, such as grouping by brands, subsidiaries, sales jurisdictions etc. So for example, retail products can be classified according to price range, purpose, manufacturer, consumer profile or essential need hierarchies. Geographic locations can be classified according to geographical region hierarchies such as zip, county, state and country, or demographic hierarchies (rural, agricultural, industrial, manufacturing and technological). The root of each hierarchy is the collection of all values for that given dimension (expressed as a wildcard "*") which in turn represents the reduction or elimination of that dimension in the Cartesian space. So the number of sales of cameras SLR-X can be expressed as Number of Sales (cameras SLR-X, *, *, *, *, *), and the grand total number of sales as Number of Sales (*, *, *, *, *, *).

These problem spaces are typical, among others, of trend and market analysis and statistical studies to find correlations, dependent and independent variables in the cause of diseases, hormonal deficiencies, etc.

3 A Relational Design

Segmentation data models typically do not exist in relational systems because segmentations would need to be bound to the primary key structure, and because of volume and performance constraints. In a relational system, space segmentation is an extension of the operational model, where segmentations correspond to unidimensional aggregation over the primary keys in the model, so if we identify products by primary key, we are likely to represent aggregations by product, and generalized segment intersections will be non-existent.

4 An HBase Distributed Parallel Design

As we design a distributed HBase architecture, one of our requirements will be segment intersection access in O(1) (i.e. in a time complexity of a single row access). Consider the following segmentations for a typical credit card product:

Credit Card Segmentations

Segmentation	Number of Values
Issuers	?
Credit card brands	5 to 10
Products	Depends on the issuer
Credit limit brackets	10
Income brackets	10
Education brackets	6
Geographic locations (zip codes)	43000
Transaction types	3
Incoming interchange types	3
Time periods	daily

The table above suggests 232.2 million segment intersections or rows per product daily. If we assumed a modest row size of 100 bytes for a segment intersection table, that would require 23.22 gigabytes of storage per product per day. If for a given issuer we assume 200 products, a five year history of segment intersections would require 8.47 petabytes of storage for that issuer. This excludes segment intersections for any of the hierarchy groups in each of the hierarchies of each segmentation.

This also remains a time-series problem simply because one of these segmentations is likely to remain time-stamps or time periods and because the operational input into the system will represent a time-series progression. The implication of this being that row key design is likely to include time-series information with appropriate prefixing for load balancing.

4.1 Persistent Layer Design

The data model represented in our HBase design can be divided conceptually in three distinct layers as follows.

- **The Operational Layer**. This layer contains the history of daily activities of the business.
- **The Segmentation Layer**. This layer contains the precomputed statistics of all the segment intersections in the market, and it is derived from the operational layer.
- **The Intelligence Layer**. This layer contains intelligence information about the business such as benchmarks, performance analysis and goals, and it is derived from the segmentation layer.

4.1.1 The Operational Layer

This layer is simply the HBase equivalent of a classical credit card relational operational model. It includes issuers, brands, products, accounts, transactions and expenses, with perhaps a minor difference in that historical state is preserved. The fact that both relational and distributed models are otherwise equivalent in content serves to showcase their differing purposes, usage and contrasting architectural design.

The relational model is populated interactively as operations, transactions, occur. The distributed HBase model is populated periodically using batch bulk loads or tools such as Sqoop.

The relational model is used primarily to retrieve isolated itemized information real time. Needless to say, the intention of the distributed HBase model is not to be able to access a specific transaction in O(1). The distributed operational model is used as the source to populate the aggregate Cartesian product of an n-dimensional Cartesian space in the segmentation layer. While for the relational model normalization is a desirable goal, for the distributed HBase model, denormalization is the norm: composite keys in the HBase model are designed to create proper load balancing, but more importantly, inter-table scan locality via synchronized region splits for efficient joins in MapReduce operations that produce these combinatorial explosions making up the Cartesian space.

Depending on a historically changing account profile, accounts will contribute to different segmentations throughout their lifetime (e.g. if the account holder's address or income bracket change in the account). In order to preserve historical state for these accounts, it is necessary to create a new account record every time an account historically-relevant state changes. To accomplish this it is sufficient to make use of HBase's time-stamped row versioning infrastructure. This way new records are created only when changes occur and the exact time-stamp of change is maintained during the construction of the segmented intersection space. Because it is likely these accounts will be accessed incrementally for progressive time-series, the time complexity of the retrieval will likely remain O(1) as only the most recent record will need to be accessed, or a worst case of O(t) where t is the number of historically significant change periods which have occurred for the account lifetime.

Operational Layer Design Comparison

Relational	Distributed
Input is real-time from interactive operations.	Input is batch bulk loading from relational import.
Output is real-time and fine-grained batch.	Output is batch via bulk MapReduce.
Purpose is the interactive retrieval of current data, financial and transactional reporting.	Purpose is to produce full historical aggregations for segment intersections.
Credit card transactions are normalized, primarily retrieved by transaction or account id.	Credit card transactions are denormalized, grouped by product, account and date for localized scans. May never be retrieved by transaction id.
Keys for different tables largely independent of each other.	Heavily composite keys for different tables provide common prefixes in order to localize inter-table relationships into common region servers to support efficient MapReduce and load balancing.

4.1.2 The Segmentation Layer

The segmentation layer is derived by MapReduce algorithms from the operational layer. The goals of this derivation is as follows:

1. MapReduce must be incremental on the time-series dimension. We want to be able to populate segment intersections for progressive time periods. This is trivial and it only means that more recent operations can be processed independently of previous operations and such operations need to be processed only once in their lifetime.
2. MapReduce must be incremental on arbitrary dimensions. We want to be able to populate segment intersections for specific segmentation values without requiring that all segment values be available. So for example, if operational values for a given issuer and product are available, we want to be able to produce all segment intersections which contain those specific issuer and product values. These segment intersections are partial because of missing values for other issuers and products.
3. MapReduce computational complexity must be O(n) on the number of operations (credit card transactions). That is, we want to process each transaction once and only once. Incremental algorithms are attractive because presumably they reduce the time complexity of their exhaustive counterparts, and thus this is stated in our goals.
4. Region splits must be distributed by credit card issuer and product sub-clusters Sub-clusters for issuers and their products can be maintained autonomously and be geographically dispersed from multi-issuer aggregation clusters.
5. Segment intersection availability to the intelligence layer must be O(1) for specific intersections and O(n) for a time-series of n periods. This means that the key for any specific segment intersection can be formulated and its row retrieved directly. This also means that the time-series for a given segment intersection will be contiguous in the HTable so that a table scan can be formulated with the exact scan range in order to avoid visiting unrelated rows. This will allow the intelligence layer to perform efficient retrieval of specific segment intersections.

With the goals stated above we achieve the following benefits:

1. A single HBase table can be used for separate issuers and separate products in isolation. Product segmentations can be updated independently when the product operational data becomes available, and in clusters localized and controlled autonomously by the owner issuer. Availability of the segmentation and intelligence layers for a given product and issuer depends exclusively on the availability of the operational data for that issuer and product.
2. Consolidated global multi-issuer segmentation and intelligence information can still be made available when and to the extent operational data from all the issuers becomes available.
3. Multi-user segment and intelligence information can be made available according to security rules, e.g. to maintain issuer and product anonymity to other issuers. Bank conglomerates can maintain anonymity between subsidiaries while allowing full visibility to conglomerate directorate for accurate supervision and management.

4.1.2.1 Populating the Segment Layer with MapReduce

In order to fully populate the segment layer (the segment intersection table) and in order to satisfy the MapReduce goals enunciated earlier, we will need three kinds of MapReduce operations. Because we understand the n-dimensional Cartesian space as leaves in the hierarchy trees for each dimension. This process consists of fully populating these hierarchy trees, where the leaves are the segment intersections in the Cartesian space. The first step consists of populating segment intersections in the hierarchy leaves from the operational layer. Subsequent steps consist of aggregating or disseminating these values throughout the rest of the hierarchical tree structures.

4.1.2.1.1 Seeding the Segment Layer from the Operational Layer

Composite keys in the HBase operational layer are designed to create proper load balancing, but more

importantly, inter-table scan locality via synchronized region splits for efficient joins in MapReduce operations that produce these combinatorial explosions making up the Cartesian space in the segmentation layer. Contrary to typical uses of the MapReduce framework, the segmentation layer output will be much larger than the operational layer input. See http://www.biomedcentral.com/1471-2105/11/S1/S15 for another example.

Because of these combinatorial explosions, the use of in-memory combiners prior to reduce operations becomes a sine qua non for a significant performance gain. The performance contributors for this layer are prioritized as follows:

1. Inter-table localization. This is the ability for inter-relating table regions to coexist locally within the same region server.
2. The use of in-memory combiners. Because of the combinatorial explosions produced by the mapper step, it is important to reduce this record explosions before they go to local server disk and before they migrate outside the region server for reduction.
3. Load balancing. Making sure these region splits in the operational layer are equal size. It is important that region splits are defined for these tables in the operational layer from creation, even when these tables are small and space constraint is not a consideration, as a way to make sure the computational load of output combinatorial explosions are evenly distributed from the start.

While subsequent aggregation from the segment intersection leaves to produce higher levels in the segment hierarchies can be aggregated as part of this single MapReduce step, it is not practical to do so because the source information from the operational layer may not be available timely so we opt to have incremental MapReduce operations that can produce these segment intersections as source data becomes available.

4.1.2.1.2 Bottom-Up Attribute Synthesis

Aggregation, such as data consolidation or totalization is a process of synthesis or bottom-up data calculation for a node in a hierarchy tree from its descendant nodes. Formulas need to be devised to perform these calculations through the reduce steps in the MapReduce algorithms, such as addition, averaging, minimum or maximum, etc. However, there are some aggregates that do not follow arithmetic formulas. For example, if the number of active accounts for a credit card product is a1, a2, …, a7 for daily periods Monday through Sunday, the number of active accounts for that week cannot be derived from the individual daily periods. For these situations, additional data structures must be devised in order to perform these computations.

4.1.2.1.3 Top-Down Attribute Inheritance

Segmentation intersections in the Cartesian space represent the leaves of the hierarchy trees and the algorithms described earlier are used to populate these segmentation hierarchies bottom up. In practice, not all data may be available for these leaves and their values may need to be produced by inheritance from their ancestor nodes. For example, certain expenses may be available for an entire issuer and not for individual products, or for monthly periods and not daily or weekly periods. For these cases, algorithms may need to be devised for top-down value distribution using mechanisms such a prorating, curve fitting and interpolation.

4.1.2.2 Incremental MapReduce

Incremental MapReduce algorithms can provide performance increases of an order of magnitude compared to cluster topology design and load balancing. They are all obviously intertwined but the anticipated design precedence is that you design for incrementality and that will drive your load balancing and distribution/cluster topology approach.

Incremental algorithms are more general when they apply to changes in existing values as opposed to just the addition on new values into the persistent state. Applications such as search engines would not exist without incremental algorithms to update word counts of changing documents. Typically, these algorithms only require the use of HBase's column versioning to obtain and propagate word count differentials, but other equally essential algorithms are far more complicated requiring the use of more elaborate supporting data structures such as dependency flow graphs, etc.

For this problem domain, a significant performance increase would arise from incremental algorithms to change existing values which have been modified or corrected (as opposed to data addition over a time-series progression), but due to the sporadic nature of these conditions for this particular problem domain, these kinds of incremental algorithms are not reviewed in this paper.

We review two approaches for incremental MapReduce algorithms regarding data addition over a time-series progression. In order for us to be able to schedule incremental MapReduce jobs we will require a scheduler that decides which segment intersections will be (partially or fully) computed during a given MapReduce job and specify the appropriate scan ranges for the job. For this we need additional data structures that depict the current Cartesian *coverage* (or progress state) of a given data set and the segment intersection table in particular. Progress state is denoted by a collection of segment intersections that represents the Cartesian space coverage provided by the data in question. Therefore, operational layer bulk imports must be submitted along with its segment intersection coverage description. So for example, bulk import coverage (issuer 453, product Visa Classic, *..., Dec 1, 2012), (issuer 453, Visa Gold, *..., Dec 1, 2012) specifies that the values to produce all the single value and wild card intersections for Visa Classic and Visa Gold for issuer 453 for the day Dec 1, 2012 are being provided by such bulk import job. When bulk imports with similar coverage for the rest of the issuer 453 products have been processed, then the scheduler in turn knows that all segment intersections for issuer 453 can be produced. A schedule job can be represented by the input and output coverage collections, where the output coverage collections are automatically derived by the scheduler. When a MapReduce job results in a coverage output of (*...) (all wildcards which symbolize the grand root of all the Cartesian space hierarchies), the segment intersection table is in a completed state.

4.1.2.1.2 Discrete Incremental MapReduce

With this approach, the calculation of a segment intersection is not initiated unless all the segmentation values for that segment intersection are already available. In other words, a segment intersection computation will not take place unless it can be fully computed at once. This is turn guarantees data integrity during multiple MapReduce jobs because this order of precedence guarantees that at most one MapReduce job is modifying a given segment intersection.

The advantage of this approach is its simplicity while the obvious disadvantage is that we are unable to obtain any intelligence information over certain (higher level) segment intersection hierarchies unless all of its contributors have been computed. In practice we may not be able to produce these higher-level segmentations because some contributors (e.g. issuers) may be habitually late with their bulk import jobs.

4.1.2.1.3 Continuous Incremental MapReduce

With this approach, the calculation of all segment intersections for which not all values are available would be partially carried out with each MapReduce job. This means that each MapReduce job will contribute to the grand root (*...) of the space. This approach requires concurrent updates of segment intersection rows. An alternative approach would be to reuse the discrete scheduling approach for contiguous incremental updates by applying new policies for segment calculation triggers other than the discrete requirement that all segment values must be available, while making sure that such segment intersection calculations are not performed concurrently for same rows and that we do not wait indefinitely for late data arrival. This scheduler would still make use of the segment intersection coverage collections to configure its MapReduce jobs and to implement its trigger policies.

4.1.3 The Intelligence Layer

With a properly populated hierarchical n-dimensional space, we are ready to apply intelligence algorithms in order to produce universal unit indicators and comparative mechanisms to support strategic decision making. See www.eknowlogie.com/CardProfit for a description of the intelligence layer for CardProfit.

5 Big Data Java Application Support

Beyond the third-party technologies readily available for the implementation of Big Data applications, such as

Apache Hadoop and HBase, there is considerable effort required in providing an inhouse application-specific infrastructure. This effort needs to cover the following areas 1) persistent layer independence, 2) persistent layer realization, and 3) generic persistent layer access and unit testing.

- Java domain object model independence. Application development investment needs to be protected from the persistent implementation technology in order to prolong the application's lifetime and reduce susceptibility to technology variations in the persistent layer. It is important that the application's domain object model can be manipulated independently of the underlying persistent implementation.
- Persistent layer realization. In order to build these technologies, we need to be mindful of the inter-disciplinary nature of Computer Science. Expertise in the following areas is desirable:
    - Incremental Graph Evaluation and Data Flow Analysis. Data dependency propagation and evaluation using graph inheritance and synthesis can be applied incrementally to provide a more significant impact in the performance of the resulting distributed application than cluster topology and load balancing, sometimes by an order of magnitude. They are all obviously intertwined but the anticipated design precedence is that you design for incrementality and that will drive your load balancing and distribution/cluster topology approach. For dependency graphs that cannot be implicitly derived from HBase tables, Neo4j can be used provided we have the guarantee that our dependency graphs will be small enough to fit in one node. The moment these graphs require sharding, it is likely that any inter-server graph analysis will bring performance to a halt, unless domain-specific simplifying assumptions about graph topology can be made to facilitate inter-server graph analysis.
    - Distributed, parallel programming. The MapReduce framework is conceptually a realization of classical distributed programming. Existing distributed algorithms are likely to be directly realizable with this framework.
    - The target technologies themselves, such as Hadoop, HBase and MapReduce.
- Generic persistent access layer and unit testing. For performance reasons, Hadoop and HBase lack facilities to manipulate content as Java types. In HBase all data manipulation is expressed in raw bytes, so it is crucial to create a rigorous, standardized schema-driven Java class organization in order to support key management and so that comprehensive automated unit test suites can be created to verify consistent byte representations for a changing object model. HBase content data structure is reminiscent of absolute binary programming in the 50's before mnemonic assembler languages were built, with one key difference, reflection, which can be used as an infrastructure for unit testing generalization and automation, provided we are rigorous in the use of these Java access layers and interfaces.

6 The Cloud Client

6.1 Cloud Client Goals

- Do we want to commit to a vendor-specific distributed parallel map reduce implementation for the life time of an application?
- Do we want to commit to a concrete vendor-specific RIA presentation framework for the life time of the application?
- Conversely, do we want to reuse across application families concrete implementation layers such as vendor-specific application-support GUI's?
- Typically, the most intrinsic and valuable asset of an application is its context-specific, domain-specific suite of business rules, and that asset is best preserved in separation from the concrete realizations of the application so that it is not washed away with the passing fads and fashion trends in the industry.

In the same way as the goals of the cloud application emphasize persistence layer independence (while relying

on map reduce algorithms in order to produce a distributed parallel cloud architecture), the goals of the cloud client emphasize presentation independence and interchangeability. Notice in particular that these goals are designed to correct the invasive permeability left by the MVC pattern which is unable to offer presentation interchangeability as that is granted only through presentation separation and client functional autonomy. These are the goals:

1. **Interchangeable Concrete Presentations**. The cloud client will interact with its concrete presentation without any knowledge of the concrete RIA implementation framework. To be specific, there will be no direct or indirect dependencies (package or class references) in the client implementation to a given concrete or vendor-specific RIA API such as GWT or JavaFX. While it may not be a design goal for many applications to support interchangeable presentations, having an architecture that supports it greatly simplifies the application by making it functionally self-contained and separate form the concrete UI layer. This eliminates the classical complexities of having complex UI widget tree navigation dictate UI component inter-relationships

2. **Functional Client Autonomy**. A corollary of the goal of concrete presentation interchangeability, is that all application functionality must reside autonomously outside the concrete presentation layer in order to avoid duplication across multiple concrete presentations. Because of MVC permeability, there is a fair share of business rules typically written directly in the concrete presentation, for example, the manner in which items from two list boxes can be interactively merged by the user. Almost invariably, rules of this nature are hard-coded in the concrete presentation, and this goal dictates that such rules be coded in the application itself. Without an architecture enforcing client-presentation separation, the concrete presentation layer's gravitational force will absorb rules that are very closely related to its operational interactivity, these rules are taken away from the application and appropriated by the presentation, making the presentation inseparable from the application. This momentum in turn forces additional gravitating business rules to be subsequently absorbed and hard coded in the concrete presentation in order to complement previously absorbed rules, thus increasingly decaying any presumption of presentation separation.

3. **Functionally-Encapsulated Presentation**. It is also desirable, in order to achieve the goals above that the concrete presentation is not given access to the client application at all, specifically, that there may not be direct or indirect references in the concrete presentation to classes and packages in the application.

6.2 Cloud Client Architecture

The proposed client architecture supporting the outlined goals consists of a functionally autonomous, self-contained *conceptual UI layer* which supports MVC, a view and controller organization with access to the client's model. This conceptual UI layer defines private interfaces that interchangeable, anonymous *concrete vendor-specific UI layers* must implement. Communication between conceptual and anonymous concrete layers is private, shielding client and concrete UI from knowledge of each other. A Spring-style dynamic factory construction of the concrete UI prevents the conceptual UI from concrete dependencies. In order to provide this shield, the conceptual UI must also rely on an application to presentation model adapter.

6.2.1 The Conceptual UI Layer

This layer describes the application presentation in terms of logical, abstracted, highly simplified components and their inter-relationships, thus characterizing the full presentation functionality which is available to the application. This layer is vendor-independent; it is void of any dependencies into the concrete UI, and therefore, it offers no actual rendering capabilities. In order for this conceptual UI to offer controller capabilities to the application, it needs to provide a vendor-independent event broadcasting model as described below.

6.2.1.1 A Doubly-Delegated Event Broadcasting Model

We propose an event broadcasting system with the following characteristics:

- *A Doubly-Delegated Event Broadcasting Model*. Design a doubly-delegated event broadcasting model, so that the *event transmitter* is the delegate of the *broadcaster* and the *event receiver* is the delegate of the *listener*, so the model must clearly identify and separate such roles. Delegation allows many to many relationships. A broadcaster can initiate multiple separate unrelated broadcasts and a listener can listen to multiple event broadcasts. Conversely, a single broadcast can be shared by multiple broadcasters to broadcast events. Other arrangements include a peer organization where all participants are broadcasters and receivers. Delegated broadcasting contrasts with the Java ChangeListener model where the broadcaster is the transmitter and the listener is the receiver, and there is a single generic transmission for any change the given broadcaster is capable of broadcasting.

- *Receivers Are Rebroadcasters*. A controller architecture can be made artificially complicated, globalized and de-objectified by the need to expose original broadcasters to receivers. This way, receivers reserve their right to privately manage events through local rebroadcasting. This also reduces the amount of unnecessary polling resulting from larger flat global listener lists.

- *Push and Pull Broadcasting*. Normally events are broadcast or pushed by a broadcaster. It is convenient for receivers to be able to individually pull events from their transmitters, for example, in situations when a receiver registers late to a broadcast.

- *Event Type Safety*. The event type can be extended (e.g. to include payload). With the use of generics for the event type, it is event type safe for the relationship between transmitters and receivers.

- *Application Autonomy*. This broadcasting model is self-contained and can be used within the conceptual UI layer and the application itself, thus promoting application autonomy without dependencies to vendor-specific presentation API's.

6.2.1.2 Application to Presentation Model Adapter

Earlier rich presentation frameworks like Swing relied on a data model of the information being rendered. Newer rich presentation frameworks, such as JavaFX and GWT, rely on direct access to an OO model of the information (via interfaces, annotations, or binding properties in JavaFX 2). While this is highly attractive from the programmatic stand point, it is unfortunate from the architectural stand point as it promotes a direct incursion of the presentation model into the definition of the application's object model. A domain to vendor-specific presentation model adapter is desirable which 1) prevents the concrete UI's access to the application's domain-specific model and 2) allows the application's model to be adapted to any chosen concrete vendor-specific presentation model. This adapter would serve to encapsulate the vendor-specific presentation. The adapter implementation options can include one or more of the following:

- *Reflection*. This is the most powerful, flexible and general of the alternatives, allowing the adapter to take arbitrary application object structures and visit them reflectively in order to populate presentation model interfaces. If necessary, directives about specific structures can be supplied to these reflective visitors via presentation-independent annotations or XML configurations.

- *Implementation of Concrete UI Interfaces*. These interfaces should be implemented within the adapter and outside the application's domain model in order to mediate two way interactions between the domain and the concrete UI models while preserving the integrity of the application model.

- *Presentation-Independent Annotations*. Application-specific, presentation-independent annotations can still be used as they will remain applicable in the adaptation to any vendor's concrete presentation UI model.

- *Object to Data Conversion*. While all the alternatives above mediate between two object models, adapters could generate simpler data structures such as JSON, and establish a contract with concrete UI's that such structures will be used to populate the UI. This is more effective when presentation widgets represent localized read-only data such as pie and bar charts.

6.2.2 The Concrete or Vendor-Specific UI Layer

This layer implements the vendor-specific components that render their conceptual UI counterparts. These components will interact with their conceptual UI counterparts through a generic rendering contract, with operations to start and stop rendering, supplying presentation models and reporting meaningful events back to their conceptual UI counterparts. Meaningful UI events captured by a concrete UI component can only be reported to the component's only visible external point of contact: its owning conceptual UI component. This conceptual UI component is then responsible to control or broadcast the event throughout its conceptual layer broadcasting system as it sees fit.

The relationship of visible conceptual and concrete UI components will be one to one. However, because the rendering process is far more complex than the description of their conceptual behavior, there will be far more concrete UI components, usually in large widget tree structures, which are built to support the rendition of every single conceptual UI component. It might be expected that the complexity of the conceptual layer, by some subjective measure, be perhaps about 20% of the concrete layer (even though the concrete layer does not implement controller functionality), which is precisely the reason for the existence of the conceptual UI layer: to serve as a simplifying contention wall to the application.

6.2.3 Go4 Design Patterns in the Client Presentation Architecture

The proposed architecture can always be revisited and rephrased in terms of well known Go4 design patterns as follows. The conceptual UI layer represents a Facade pattern first, and a Bridge pattern second. It is essentially a Facade because the main objective of the conceptual UI is to simplify the presentation model of the application. It is also a Bridge because in accomplishing the first goal, it allows us to abstract the concrete or vendor-specific implementation details and make vendor implementations interchangeable. Injection of a vendor-specific UI is made anonymously by the conceptual UI using a Spring-style Factory pattern. The application-presentation model adapter makes use of the Adapter pattern because the design intention is to maintain full separation of the application and presentation models as they are manipulated by their respective clients.

6.2.4 A Brief Example of the Conceptual and Concrete UI Architectures

The conceptual and concrete UIs are implementations of the *UIComponent* and the *Renderer* interface hierarchies respectively. The conceptual UI implements UIComponents, and the concrete UI implements vendor-specific Renderers. Each UIComponent class must specify a static *UIComponentID* to identify the component by a property name and a collection of renderer types which can be used as concrete renderers for this component. A *RendererType* is not part of the Renderer hierarchy, it is just a marker class used to enumerate renderers according to certain distinguishable rendering characteristics. Similarly, each Renderer class must specify a static *RendererID*, in order to identify the renderer by a property name and the renderer type or types it implements, along with supported layout characteristics such as orientations (vertical, horizontal, square).

So for example, a UI that displays a list of numeric values may use a UI component ID that specifies the renderer types which may be used to display such a list, and which may include a *TABLE_RENDERER_TYPE*, a *NUMBERS_LIST_RENDERER_TYPE*, and a *LIST_DISTRIBUTION_RENDERER_TYPE*. In turn, renderers that implement pie charts and one dimensional bar charts can include the *NUMBERS_LIST_RENDERER_TYPE* in their ID's. Similarly, a bell distribution curve renderer can also be used to display this list of numeric values according to their statistical dispersion. In turn, the concrete UI can offer menus to select among all the rendering options available for a given UI component in display. The UI component ID can also offer a same ID event broadcaster so that these menu-driven interactive rendering decisions can be made collectively for all instances of UI components with same ID.

A *RendererRegistry* is a singleton where all available renderers ID's are posted for use. For a given UI component, the registry can then identify the default renderer, as well as all the renderers available for display. The renderer mapping and selection process in the the registry belongs to the conceptual layer, it is vendor-independent, but the registry population is done by the concrete layer. The conceptual and concrete layers are also stratified vertically, with generic libraries at the bottom, and application or domain-specific libraries on top. A

Spring bean factory can be used to instantiate the registry, renderers and special-purpose renderer configurators.

A *UILayout* is an extension of the UIComponent hierarchy that has the capability to control the visibility of a group of components. To the extent that these components are displayed for simultaneous viewing, a layout must control the spatial arrangement so that they can indeed be visualized simultaneously. Layout components in the conceptual UI will typically remain very abstract and succint, confined to specifying the list of components that need to be displayed simultanously. UI components are not referenced directly by these layouts. They are referenced though *UIComponentHolders* which hold the UI component in question and provide contextual usage information, such as whether the component is optional or required, whether it can be minimized into a layout tray or maximized. This way, UI components are reusable by multiple layouts and other components. A UIComponentHolder or context can hold one and only one UI component, but a UI component can be held by several holders. Similarly in the concrete layer, a renderer cannot be included directly into a concrete GUI or widget tree (e.g. a JavaFX or GWT component tree). Instead, it always has to be held inside a *RendererHolder* with equivalent structure: a renderer holder may hold one and only one renderer but a renderer may be held by multiple renderer holders. This way, components can be shared by multiple layouts, and these layouts can be switched on the fly without affecting the other layouts. This architecture lends itself for the implementation of an Eclipse-style tile layout with movable tile boundaries, tile tray minimization and maximization.

In the concrete UI, renderers and layouts may be implemented using anonymous arbitrarily complex widget tree structures. The conceptual UI serves to exercise full control over the concrete UI on application-related issues while separating itself from these overly complex concrete widget tree implementations.

A. References

[DeCandia07] DeCandia, G., et.al. (2007). Dynamo: Amazon's Highly Available Key-Value Store. ACM Symposium on Operating Systems Principles. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[George11] George, Lars (2011). HBase: The Definitive Guide. O'Reilly

[Lakshman09] Lakshman, A., Malik, P. (2009). Cassandra – A Decentralized Structured Storage System. http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf

[Lamport78] Lamport, Leslie (1978). Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21 (7). http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf