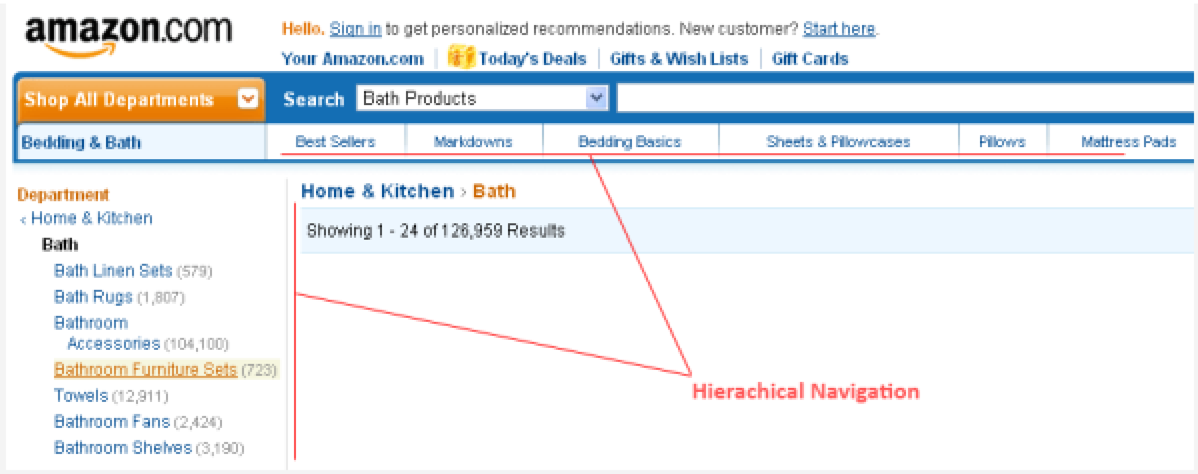
Hierarchical Navigation and Faceted Search on Top of Oracle Coherence

Some time ago I participated in design of a backend for one large online retailer company. From the business logic point of view, this was a pretty typical eCommerce service for hierarchical and faceted navigation, although not without peculiarities, but high performance requirements led us to the quite advanced architecture and technical design. In particular, we built this system on top of Oracle Coherence and designed our own data structures and indexes.

In this article, I describe major architectural decisions we made and techniques we used. This description should not be considered as a solid blueprint, but rather a collection of the relatively independent ideas, patterns, and notes that can be used in different combinations and in different applications, not only in eCommerce systems.

Business Logic: Hierarchical and Faceted Navigation

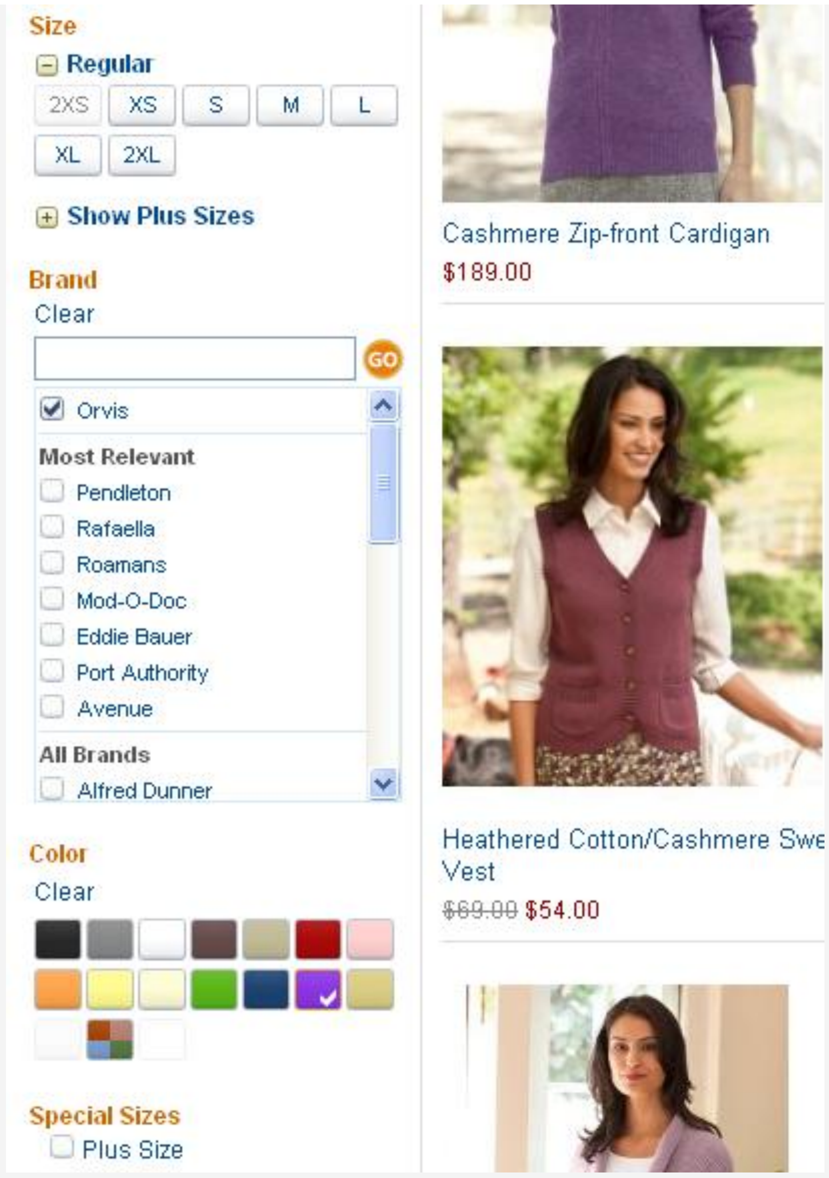
I cannot disclose customer's name, so I will explain business logic using *amazon.com* as an example, fortunately the basic functionality is very similar. The first piece of functionality is structural or hierarchical navigation through categories and products, which are the main business entities of the system. Categories are organized in a tree-like structure and the user is provided with several controls that enable him to navigate through this tree starting from the highest categories (like departments on *amazon.com*) and going to the lowest ones:


Hello. Sign in to get personalized recommendations. New customer? Start here.

Showing 1 - 24 of 126,959 Results

Hierarchical Navigation on Amazon.Com

Each product can be explicitly associated with one or more categories of any level and category contains a product if this product is explicitly associated with it or associated with any of its subcategories. These structural dependencies between categories and products are relatively static (the system refreshes this information daily), but operations team can change separate relations in runtime to fix incorrect data or to inject other urgent changes. Besides this, each product has some transient information like in-stock availability that is a subject of frequent updates (every 5 minutes or so).
The second important piece of functionality is a faceted navigation. Categories can contain thousands of products and user cannot efficiently search though this array without powerful tools. The most popular way to do this is a faceted navigation that can be thought as a generation of dynamic categories based on product attributes. For example, if the user opens a category that contains clothes, products will be characterized by properties like size, brand, color and so on. Available values of these properties (called facets) can be extracted from the product set and shown on the UI to enable the user to apply *user-select filters,* which are particular AND-ed or OR-ed combinations of the facet values:



Faceted Navigation on Amazon.Com

Each facet value is often accompanied with *cardinality*, i.e. number of products that will be in the results set if this filter is applied. When user clicks on a facet, the system automatically applies the selected filters and narrows the product set according to the user interests. It is important that this style of navigation assumes high interactivity – each selection leads to recomputing of all available facets, their cardinalities, and products in a result set.

There is a lot of information about faceted search on the web. I can recommend this article by Peter Morville and Jeffrey Callender for further reading. We will also return back to some details of business logic in the section devoted to implementation of the faceted navigation.

From the backend perspective, hierarchical and faced navigation requires the following operations to be implemented:

- **getProductsAndFacets(CategoryID, UserSelectedFilters)** – return all products within the category filtered in accordance with the user-selected filters, compute available facet values and corresponding cardinalities for the filtered product set.
- **traverseCategoryHierarchy(CategoryID)** – return ancestors and descendants of the given category in the tree of categories. Depth of traversal is specified by the frontend.
- **getProducts(ProductID[])** – return a product domain entity that contains product attributes, prices, images etc. This information is used to populate a page with product and display product details.
- **getCategories(CategoryID[])** – return a category domain entity that contains category attributes and properties.
- **getProductsTransientAttributes(ProductID[]),getCategoryTransientAtributes(CategoryID[ ])** – return a short list of attributes that are the subject of frequent changes (the in-stock availability etc.) The rationale behind these methods is that frontend should be able to fetch transient information very efficiently and separately from fetching of heavy-weight domain entities because this information cannot be cached.

System Properties and Major Technical Requirements

From the technical perspective, the following properties should be highlighted:

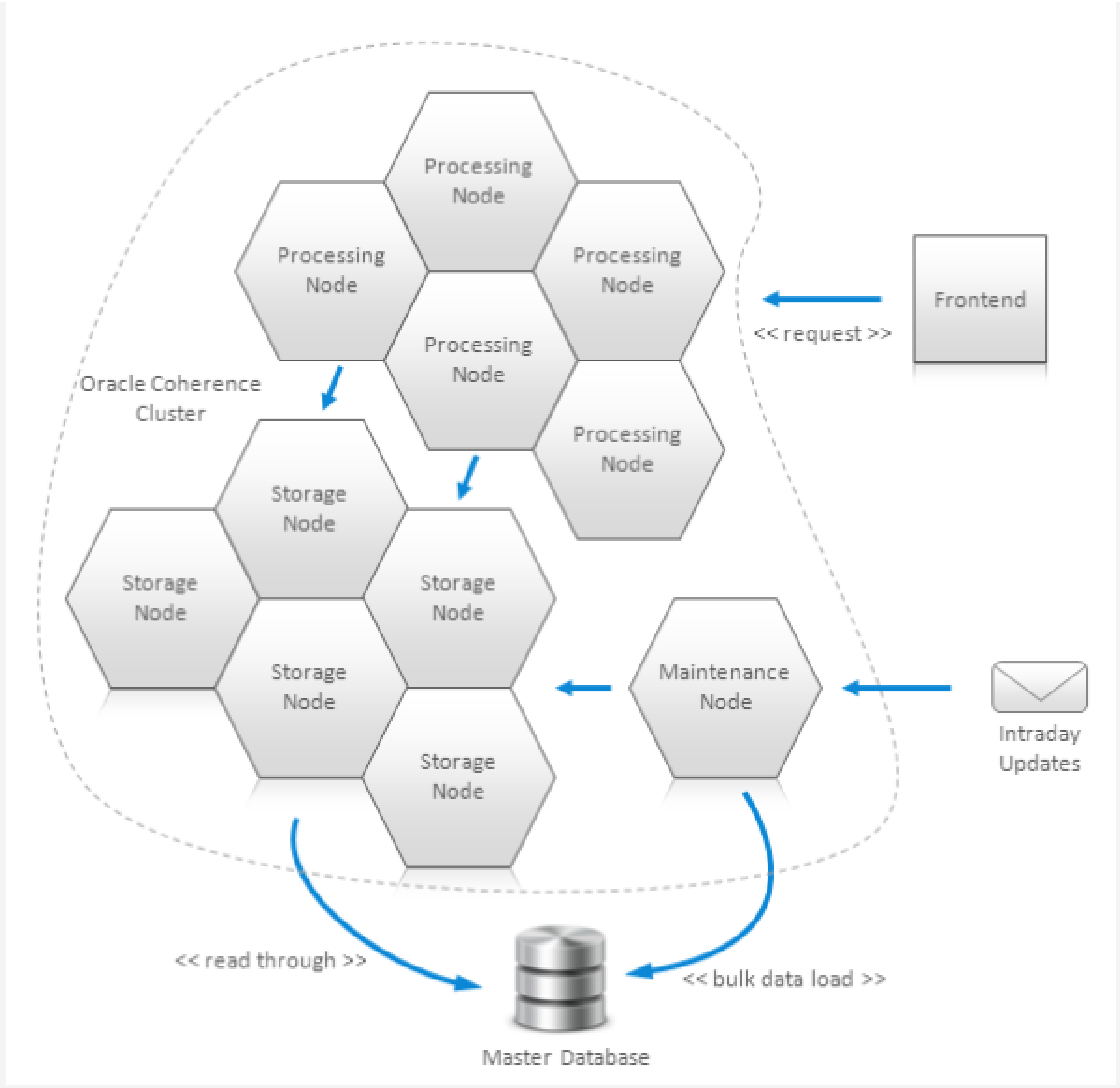
- All data is initially stored in the relational database, but this database is heavily loaded because it is a master record for many applications. So, the only way was to **cache all necessary data** to minimize interaction with RDBMS.
- The content that is delivered to users (categories and products) is pretty much static. In such cases, content delivery network (CDN) is typically used to cache majority of the content and shield the system from high workload. Nevertheless, there were two obstacles that decrease efficiency of CDN in this project:
  - Faceted navigation leads to a high amount of different views because users are able to select arbitrary combinations of facets, and, consequently, many unique requests should be served.
  - Product in-stock availability is transient, especially for the certain periods of eCommerce system life cycle (sales and so on). This means that content – products and facets – is sporadically updated every few minutes.
- Taking into account the previous considerations, performance requirements were set as**1000 faceted navigation requests/second** per typical hardware blade.
- Data capacity of the system is not less than **1 million products.**

- Structural data are completely reloaded from the RDBMS every night. Transient information updates and requests for minor changes of structural information can arrive every few minutes.
- The system is implemented in Java.

Deployment Schema and High-Level Architecture

The major architectural decision was to use in-memory data grid (IMDG) to shield the master RDBMS from workload during request processing. Oracle Coherence was chosen as an implementation. Coherence is used as a platform that provides distributed cache capabilities and can serve as a messaging bus for coordination of all application-level modules on all nodes in the cluster.

The deployment schema includes three types of nodes - processing nodes, storage nodes, and maintenance nodes. Processing nodes are responsible for requests serving and act as Coherence clients. Storage nodes are basically Coherence storage nodes. Maintenance nodes are responsible for data indexing and processing of transient information updates. Both Storage and Maintenance nodes do not serve client requests. This deployment schema is shown in the figure below:



Deployment Schema

Nodes can be dynamically added or removed from the cluster. All nodes (processing, storage, maintenance) host the same application that contains all modules for request processing, maintenance operations, and Coherence instance. Basically, deployments on all nodes are identical and can serve both client requests and maintenance operations, although each type of nodes has its own configuration parameters. The rationale behind this architecture can be recognized as a pattern:

Pattern: Homogeneous Cluster Nodes

Problem

There is a clustered system that consist of multiple business services and auxiliary modules (data loaders, administration controls, etc). The deployment process is going to be complex if each module is deployed as a separate artifact with its own deployment schema and configuration.

Solution

Different groups of nodes in the cluster can have different roles and serve different needs, but it may be a good idea to create one application and one artifact that will be deployed throughout the cluster. Different modules of this application are activated on different nodes depends on explicitly specified configuration (say, property files) or just because of usage pattern (say, certain requests are routed only to particular nodes).

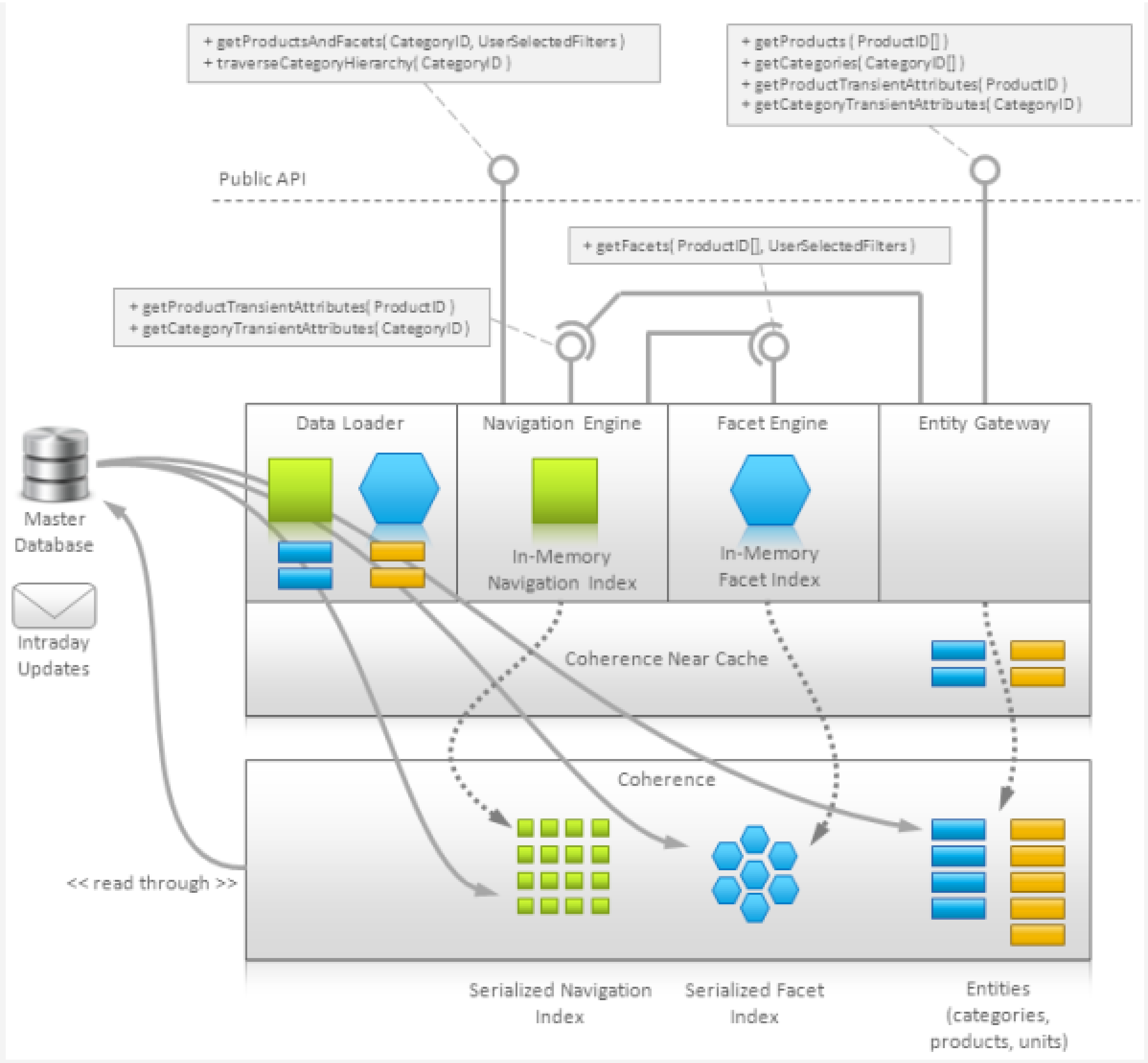
Results

This approach simplifies deployment and release processes, mitigates risk of incorrect deployment or misconfiguration. Development and QA processes are simplified because one can use either singe node or multiple nodes to run fully functional environment.

Turning to the internals of the application itself, we can see that it includes the following components (these components are depicted in the figure below):

- Data Loader. The first role of this component is to fetch data from the master DB, assemble domain entities, and push these entities to Coherence. The second role is to build navigation indexes (these indexes will be described in the further sections), split them into chunks, and flush to Coherence. The rationale behind splitting into chunks is that indexes can be quite large (hundreds of megabytes), and Coherence is not intended for storing of such large entities, transmission of these entities can block Coherence network IO and crash the cluster. The third role of the Loader module is to receive intraday updates and apply patches to the indexes and domain entities.
- Entity Gateway. The role of this module is to return information about particular entities, products and categories. Basically, this module is just a facade for Coherence. It takes domain entities from Coherence, compute fields that depend on transient information using navigation index, and return data to the client.
- Hierarchical Navigation Engine. This engine is responsible for hierarchical navigation and works as a primary navigation service for external clients. Besides this, the navigation index is a master record for transient attributes, so other modules like Entity Gateway request these attributes from the Navigation Engine. Implementation of the engine will be described in the next section.

- Facet Engine. This engine is responsible for computation of facets and for filtering according to user-selected filters. Implementation of this module will be discussed later.



+ getProductsAndFacets(CategoryID, UserSelectedFilters)
+ traverseCategoryHierarchy(CategoryID)
+ getProducts (ProductID[])
+ getCategories(CategoryID[])
+ getProductTransientAttributes(ProductID)
+ getCategoryTransientAttributes(CategoryID)
Public API
+ getFacets(ProductID[], UserSelectedFilters)
+ getProductTransientAttributes(ProductID)
+ getCategoryTransientAttributes(CategoryID)
Data Loader
Navigation Engine
Facet Engine
Entity Gateway
Master Database
In-Memory Navigation Index
In-Memory Facet Index
Intraday Updates
Coherence Near Cache
Coherence
<< read through >>
Serialized Navigation Index
Serialized Facet Index
Entities (categories, products, units)

Component Diagram and Data Flows

Data Loader is active only on the Maintenance nodes where it has a plenty of resources for temporary buffers, index compilation tasks and so on. All updates and indexing requests are routed only to the Maintenance nodes, not to Processing/Storage nodes. Such separation of data loader and other maintenance units can be recognized as a common pattern:

Pattern: Maintenance Node

Problem

There is a cluster of nodes where each node is able to serve both business and maintenance requests. Maintenance operations can consume a lot resources and impact performance of business requests.

Solution

Maintenance operations like data indexing can be handled by any cluster node when a distributed platform like IMDG is used. Nevertheless, it is often a good idea to use a dedicated node for this purpose. This node can be identical to other nodes from the deployment point of view (the same application as on the other nodes), but user requests are not routed to it and more powerful hardware can be used in some cases.

Results

On the one hand, maintenance node provides potentially resource-consuming indexing processes with dedicated hardware capacities. On the other hand, maintenance processes do not interfere with user requests.

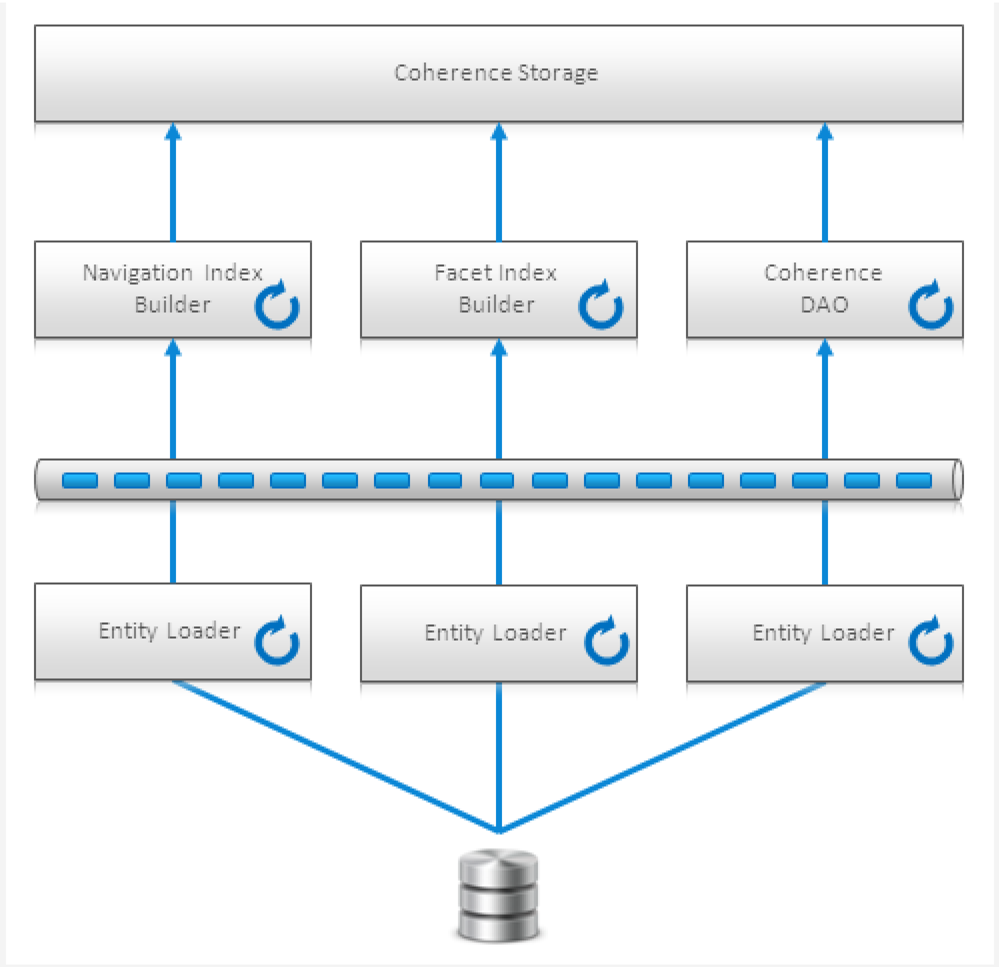
Data Loader loads *all active* data to Coherence during each daily update, but there is "*dark matter*" that is not loaded into Coherence but occasionally requested by some clients. For instance, this matter is obsolete products and categories that are not visible on the site and not available for purchase. Coherence Read-Through feature is used to cope with these entities - it is acceptable to load them from the RDBMS on demand because the number of such requests is very low.

Implementation of Data Loader

Design of Data Loader is influenced by two major factors:

- Loader should efficiently fetch and process large data set in a relatively short time.
- There are multiple consumers like index builders or entity saves that should process the same data.

As a result, Data Loader is organized as an asynchronous pipeline (Pipes and Filters design pattern) where batches of entities are loaded from RDBMS by a set of units that work in parallel threads. Loaded entities are submitted to a queue, and each consumer works in its own thread taking batches and processing them independently from the other participants. This schema is shown in the figure below:



Data Loading Pipeline

This schema is relatively simple because there is only one data source and structure of entities is not too complicated. Nevertheless, this pipeline can become more complex if there are multiple data sources and one business entity is assembled using several sources. In this case, a batch of entities can be initially loaded from a single source and then passed to another loader that enriches entities by additional attributes and so on.

Pattern: Data Loading Pipeline

Problem

A system should be populated with a large data set that come from single or multiple sources. One business entity can depend on multiple sources. There are many consumers of the loaded business entities that index, persist, or process entities.

Solution

Adopt the Pipes and Filters pattern. Implement each operation (loading or indexing) as an isolated unit that produces or consumes entities. Data producers or loaders should be driven by incoming requests that specify data to be loaded. Connect all units via asynchronous data channels and run multiple instances of each unit as an independent process.

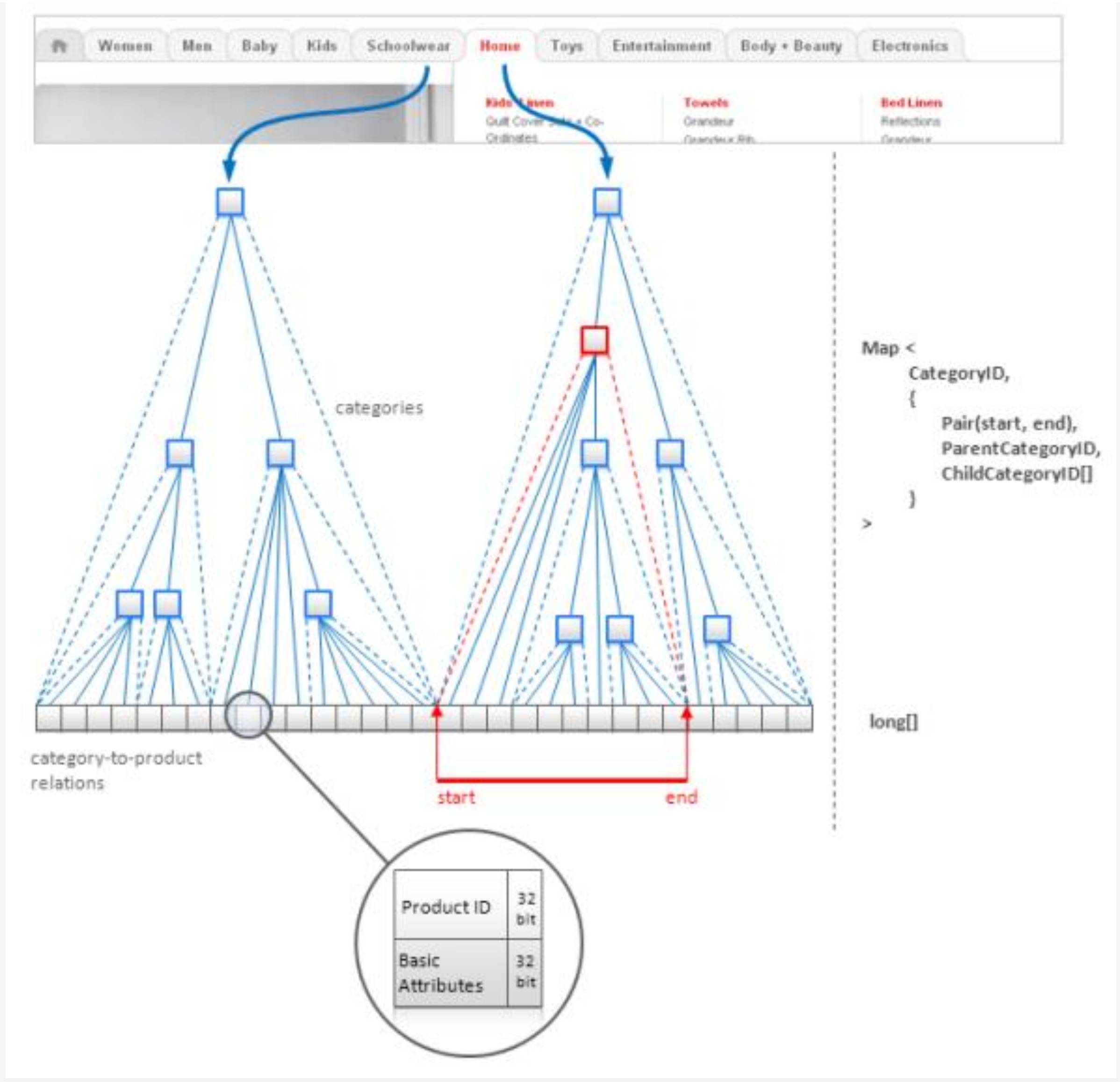
Results

Data Loading Pipeline allows one to organize efficient data loading in a multithreaded environment. All units can work in a batch mode, and more parallel instances can be easily added. A special attention should be paid to the memory consumption because queues with entities can consume a lot of memory if a system is not balanced or misconfigured.

Data inconsistency during saving of new data to Coherence is practically avoided using techniques that were described in one of my previous articles.

Implementation of Hierarchical Navigation

When we first started to work on the navigation procedures, we first tried to do it using standard Coherence capabilities, i.e. filters and entry processors. This attempt was not very successful from the performance point of view due to high memory consumption and relatively low performance in general. The next step was to design a compact data structure that supports very fast category tree traversal and extraction of products by Category ID. The structure we created is based on thenested set model, it is shown in the figure below:



Hierarchical Navigation Index Structure

A navigation index represents a huge array of product IDs and their basic attributes that are frequently used in computation and filtering, for example, in-stock availability. In our domain model these attributes are binary, hence we efficiently packed them into integer numbers where each bit is reserved for a particular attribute. Each element of this array corresponds to the product-to-category relation and one product ID can occur in this array multiple times if product is associated with multiple categories. Hierarchy itself is stored as an indexed tree of category IDs and each node contains two indexes in product-to-category array. This indexes point to start and end positions of relations that belong to the particular category.

The second notable feature of this navigation solution is that each Processing Node fetches index from Coherence and entirely caches it in local memory. This allows one to perform navigational operations without touching heavy-weight domain objects. If data volume becomes high, it is possible to partition index into several shards and perform distributed processing, although it was not a case in our application (index with millions of products can be easily handled by one JVM). This technique can be considered as a common pattern (or anti-pattern, it depends on scalability requirements):

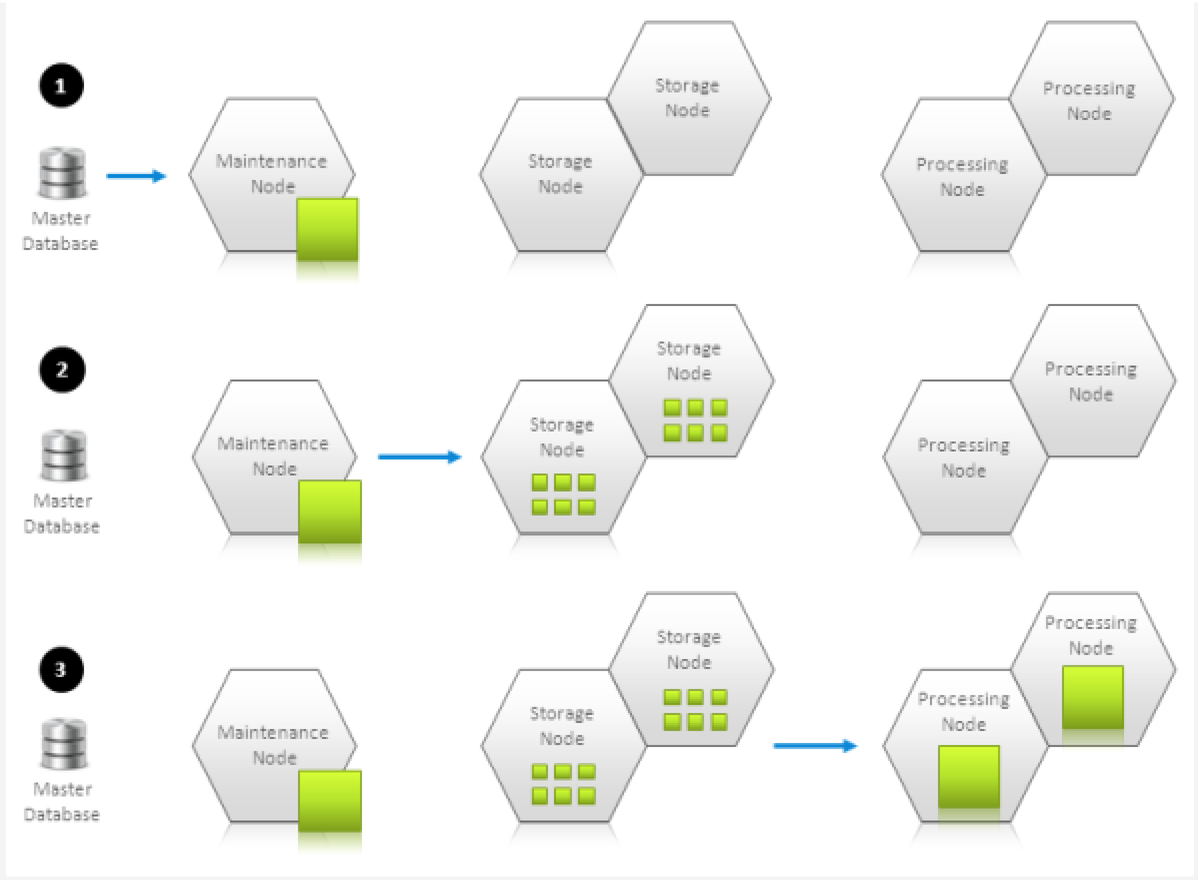
Pattern: Replicated Custom Index

Problem
There is an application with a distributed data storage. It is necessary to perform a special type of query that involves limited amount of attributes for each entity, but complex business logic or high performance requirements make standard distributed scans inefficient.

Solution
When a non-standard traversal or querying is required and amount of involved data is limited, each node in the cluster can cache domain-specific index and use it to perform the operation.

Results
This approach can be very efficient when standard indexes do not work well, but it can turn into scalability bottleneck if implemented incorrectly. If there are reasons to assume that index will become too large to be cached on one node, this is a serious argument against this approach.

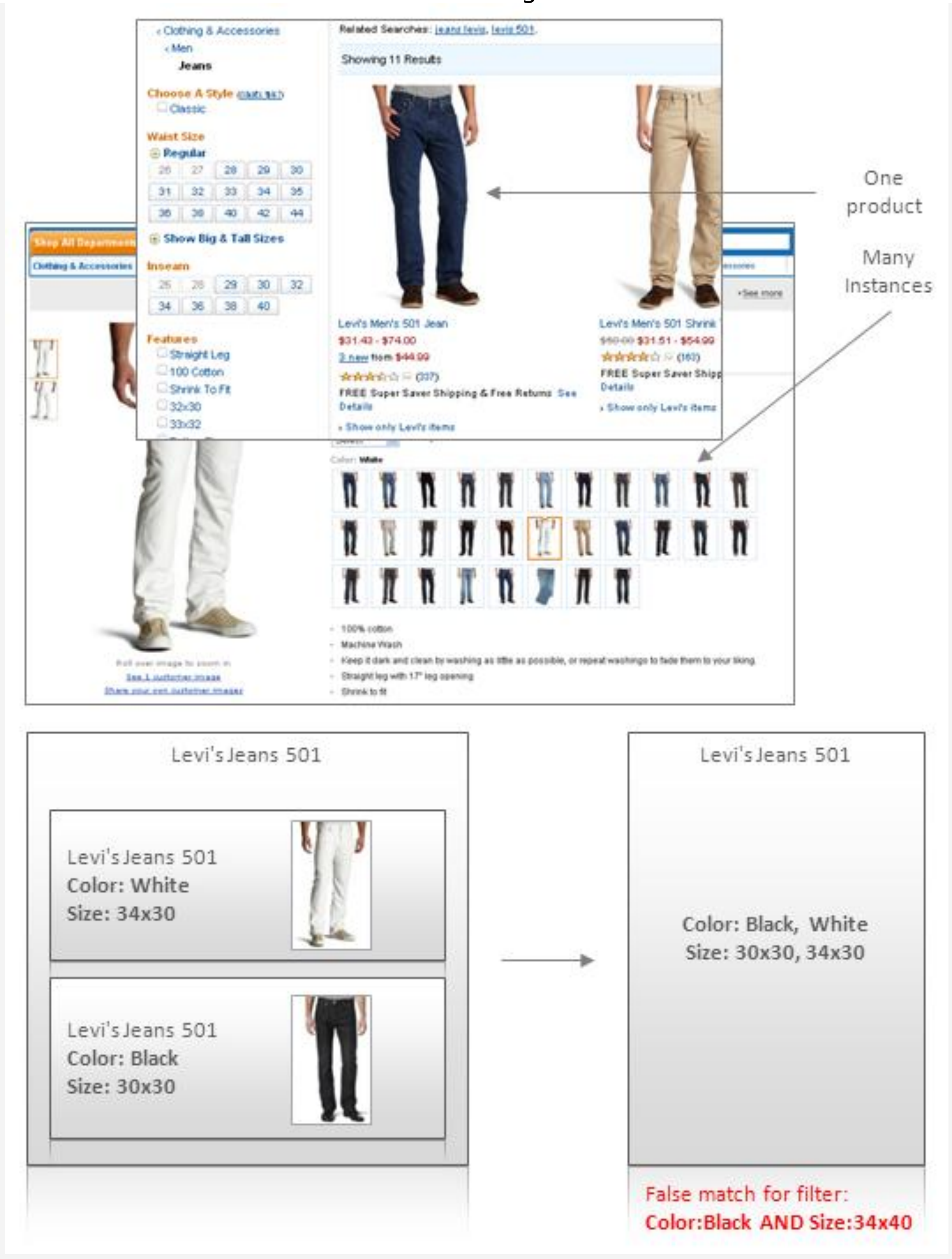
Index propagation throughout the cluster is shown in the figure below. Maintenance Node loads data from the Master DB, builds index, saves it in a serialized partitioned form to Coherence, and then Processing Nodes fetch it and cache locally:



Index Building and Propagation

Implementation of Faceted Navigation

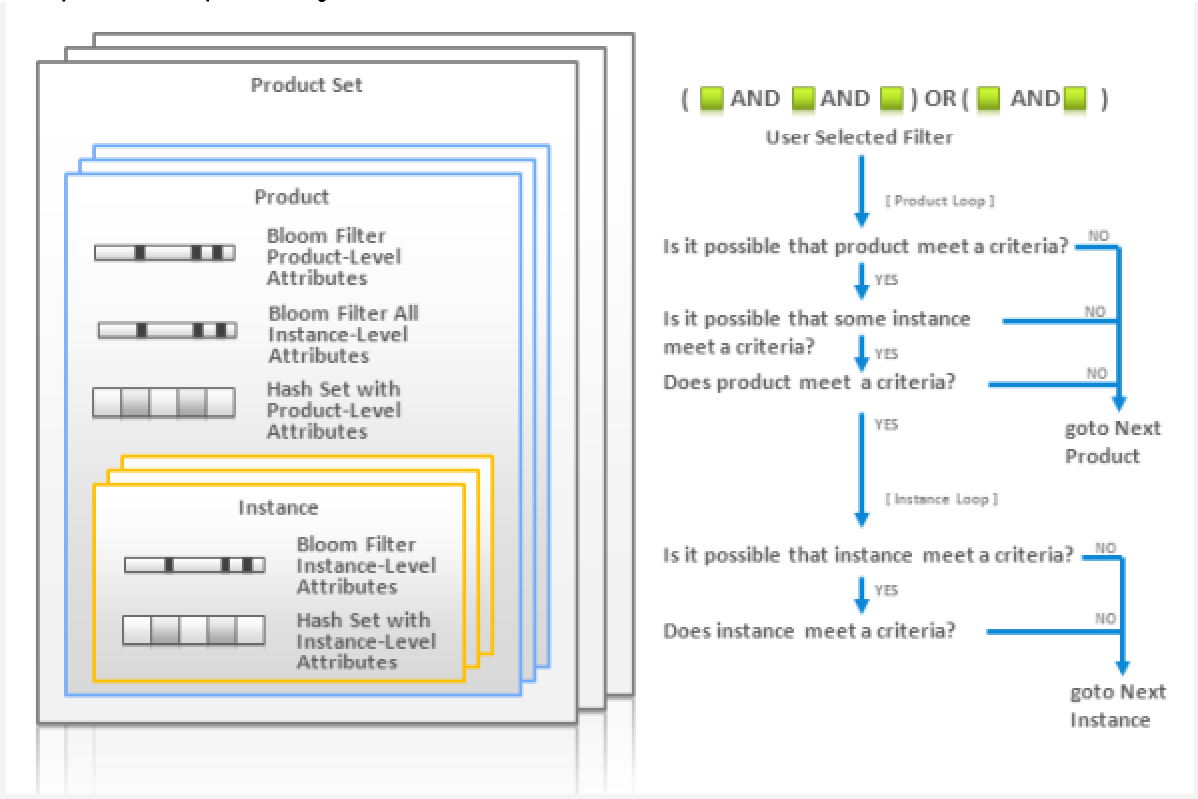
Faceted Navigation was described in the first section of this article, but it should be mentioned that logic of computation is not always straightforward, but often affected by business rules and peculiarities of a business model. As an interesting example, we can consider the following use case. Imagine that, according to the business model, product is not a final item of purchase, but a group of such items. For instance, when user looks into the Jeans category, he or she can see Levi's Jeans 501 as a product, but the actual item to be purchased is a particular instance of Levi's Jeans 501, say Levi's Jeans 501 of size 34×30, white color. Considered as a product domain entity, Levi's Jeans 501 will contain many particular items of a different color and size. From the faceted navigation perspective, this leads to the interesting issue. At the first glance, it is fine to attribute each product with all sizes or colors that can be found among all its instances and build facets based on this information. Now imagine that there are two instances of Levi's Jeans 501 – one is of size 34×30 and in white color, another one is one is of size 30×30 and in white color. If the user looks for black jeans of size 34×30, this product will match the filter if it is simply attributed by a plain list of instance-level attributes. Nevertheless, there are no black jeans of size 34×30 in the store. This situation is illustrated in the figure below:



Incorrect Modeling of Products and Instances

This is just a one example of non-trivial issues with facetization logic. Many more issues and merchandiser-driven tweaks can appear in a real system. The conclusion is

that faceted navigation can be pretty sophisticated and certain implementation flexibility is required.
To cope with such issues, it was decided to keep the design of a facet index very straightforward and do not use data layouts like inverted indexes. Basically, all products, their instances and higher level groups of items are stored just like nested arrays and maps of objects:


Product Set
Product
Bloom Filter Product-Level Attributes
Bloom Filter All Instance-Level Attributes
Hash Set with Product-Level Attributes
Instance
Bloom Filter Instance-Level Attributes
Hash Set with Instance-Level Attributes
(🟩 AND 🟩 AND 🟩) OR (🟩 AND 🟩)
User Selected Filter
[Product Loop]
Is it possible that product meet a criteria?
NO
YES
Is it possible that some instance meet a criteria?
NO
YES
Does product meet a criteria?
NO
YES
goto Next Product
[Instance Loop]
Is it possible that instance meet a criteria?
NO
YES
Does instance meet a criteria?
NO
goto Next Instance

Facet Index

All attributes are mapped to the integer values and these values are compactly stored in open addressing hash sets inside each instance or product. This allows one to iterate over all items within a category, efficiently applying user selected filter to each item, and increment facet counters for all attributes that are inside accepted items. I provided a detailed description of data structures and algorithms that allow one to do this in my previous post.
If the user selected filter includes many attributes it may be inefficient to check all these attributes one by one for each item. Performance of filtering can be improved using Bloom filter that allows one to apply a filter of several terms to a set of attributes using a couple of processor instructions. Bloom filter is liable to false positives, so it can not completely replace traditional checks using hash sets with attributes, but it can be used as a preliminary test to decrease a number of relatively expensive exact

checks. This technique is used in a number of well-known systems, Google Big Table and Apache HBase are among them.

Pattern: Probabilistic Test

Problem

There is a large collection of items (domain entities, files, records etc). It is necessary to provide the ability to select items that meet a certain criteria – simple yes/no predicate or complex filter.

Solution

Items can be grouped into buckets. Each bucket contains one or more items and has a compact *signature* that allows one to answer the question "*is there at least one item inside the bucket that meets the criteria*". This signature is typically a kind of hash that has much smaller memory footprint than the original collection and liable to false positives. Query processor tests bucket's signature and, if results shows that bucket potentially can contain the requested items, it goes into the bucket and checks all items independently.

Results

Probabilistic testing is good to trade time to memory or IO to memory. It increases memory consumption because of signatures, but allows one to significantly decrease volume of processed data for selective queries.

Replicated Custom Index pattern is used to distribute Facet Index throughout the cluster, just like Navigation Index.

Conclusions

The described design showed the following properties after being in production for a long time:

- (+) Computational performance is superior in comparison with the general-purpose databases and third-party products.
- (+) The deployment schema is very efficient at all stages of development, functional testing, performance testing, and production maintenance because of its simplicity and flexibility.
- (+) Cost of ownership and development is pretty low in comparison with third-party products usage due to high flexibility and relative simplicity of the used data structures.

- (–) Scalability by data is not a built-in feature of the described design because of non-sharded replicated indexes. Nevertheless, actual capacity is relatively high for eCommerce domain and sharding capabilities can be added.
- (–) In the long term perspective there is a negative tendency to over-complicated extensions around the core structures that are caused by complication of the business logic.

Source: http://highlyscalable.wordpress.com/2012/04/02/architecture-of-high-performance-ecommerce-backend/