Informe TP Big Data

Procesamiento batch, streaming y visualización

Agustin Marseillan
Martin Sturla
Maria Victoria Mesa Alcorta

Funcionalidades implementadas

Se contaba con información de una empresa proveedora de servicios de TV por cable. Por un lado se tenía información acerca de la grilla (separada en canales y programación de cada canal) y de los clientes. Por otro lado, información en colas de mensajes que representaban el comportamiento de los clientes. Los posibles mensajes eran:

1. Cambiar el canal:
    - Id del cliente
    - Canal
    - Hora
2. Prender la TV:
    - Id del cliente
    - Información que indica que la acción fue prender la TV
    - Canal
    - Hora
3. Apagar la TV:
    - Id del cliente
    - Información que indica que la acción fue apagar la TV
    - Hora

A partir de toda esta información, se realizaron ciertas implementaciones que permitieron sacar métricas y finalmente gráficos para la visualización de la misma.

Se obtuvieron métricas de tipo batch y tiempo real.
Las métricas de batch calculadas son:

- Anuncios publicados por cada canal
    - Se consideró como la cantidad de anuncios publicados en un canal, total.
- Top 10 de canales en audiencia
    - Se consideró “audiencia” a cada persona que visitó el canal, sin importar por cuánto tiempo. Cada persona fue contada sólo una vez.
- Top 10 de categorías en audiencia
    - Exactamente las mismas consideraciones que la métrica anterior. Cabe destacar que si una persona visitó dos canales que comparten una categoría, cuenta como sólo una visita para cada categoría.
- Promedio de tiempo dedicado por televidentes a cada canal por día
    - Para esta métrica, se calculó por cada día en particular, la cantidad de gente que visitó cada canal, y se dividió el total de tiempo visto por todos los televidentes por la cantidad total de ellos. Es decir si para el canal 1 el cliente 2 y 3 lo miraron 3 minutos, y el cliente 4 lo miró 6 minutos en 3

sesiones de dos minutos, el valor para ese día será de 4 minutos. Cabe destacar que cada “vista” está indexada por día según el comienzo de ella. Es decir, si un usuario prendió un canal a las 23:58 el 5 de abril y cambió de canal a las 00:02, se suma 4 minutos al 5 de abril.

- Promedio de tiempo dedicado por televidentes a cada categoría por día
    - Exactamente las mismas consideraciones que el punto anterior, pero para categorías.
- Audiencia por tipo de cliente
    - Se consideró “audiencia” a cada persona que miró televisión, sin importar por cuánto tiempo. Cada persona fue contada una sola vez.
- Audiencia por grupo familiar
    - Las mismas consideraciones que la métrica anterior, pero agrupando por grupo familiar.
- Los peores shows
    - En esta métrica se calculó para cada show la cantidad de personas que cambiaron de canal.

Para las métricas de tipo tiempo real, minuto a minuto de la última hora, en caso de que un televidente cambiara de canal en ese mismo minuto, se tuvo en cuenta el solo último canal al que cambió.
Las métricas son:

- Televidentes por canal
- Televidentes por categoría
- Televidentes por grupo familiar
- Televidentes por tipo de cliente
- Televidentes total

Luego de calcular esas métricas, se implementó un dashboard en el cual se muestran gráficos para poder visualizar los resultados obtenidos.

Tecnologías utilizadas

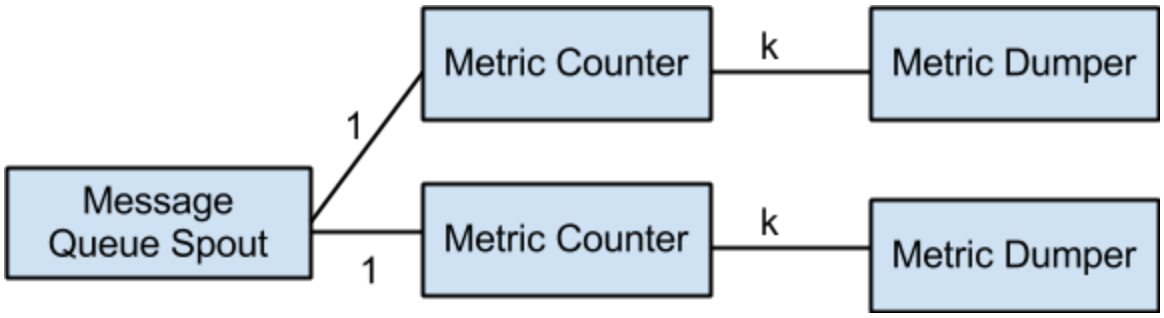
Batch

Para realizar las consultas, se utilizó la plataforma Pig, la cual permite el procesamiento de grandes cantidades de datos semi-estructurados utilizando Hadoop MapReduce.
Se eligió Pig por su mayor madurez en desarrollo, es decir su librería estándar más extensa y útil. Entre otras, se utilizaron las funciones TOKENIZE, UnixToIso, y muchas otras para operar sobre strings. Además, el hecho de poder declarar UDFs propias que devuelvan estructuras de arbitraria complejidad resultó muy útil a la hora de cargar datos y efectuar consultas más complejas.

Tiempo Real

Se utilizó Storm para java para el desarrollo de las topologías que realizan las métricas y mySql para guardar dichos resultados.
Se optó por una base de datos con tablas de schema (metric_key:String, mite:Long, quantity:Long) en la que se guardará cada métrica obtenida por minuto, en donde *mite* es la cantidad de minutos transcurridos desde la época (es decir tiene un timestamp asociado que representa un minuto entero). El campo *metric_key* es la clave de la métrica. Por ejemplo para las métricas que tratan con canales, es el nombre del canal. La *quantity* no es más que la frecuencia de la clave asociada. Se pensó de esta manera para facilitar la futura implementación de una visualización, así como la extensibilidad para la generación de nuevas métricas; sin embargo esto último se explicará más adelante.

Topología



Como se puede apreciar, existen tres componentes en la topología, los cuales son detallados a continuación:

Message Queue Spout: Un spout que se encarga de leer desde una cola de ActiveMQ, separar el json en sus campos respectivos, y emitirlos como una tupla.

Metric Counters: Los metric counters se encargan de recibir las tuplas emitidas por el spout y actualizar una tabla de frecuencia. Cuando reciben un timestamp que no pertence a la franja de un minuto que estaban procesando, emiten su tabla de freuencia en una cantidad variable de tuplas (según cuantos registros haya), y luego continúan actualizando su tabla. El paralelismo de estos bolts se seteó a 1. Las razones para que llevaron a tomar esta decisión están detalladas en la sección de problemas. Estos bolts requieren una función de obtención de característica, *charFunction*, que era explicada más adelante.
Metric Dumpers: Los Metric dumpers se encargan de recibir las tuplas emitidas por los Metric Counters y persistirlas en una base de datos mysql a través de jdbc. A pesar de que esto podría haber sido hecho en el bolt anterior, se decidió separar para evitar que el los Metric Counters hagan operaciones bloqueantes, y por prolijidad. Debido a que las bases de datos son concurrentes, estos bolts son fácilmente paralelizables. Estos bolts pueden tener una *mapFunction*, que será explicada más adelante.

Visualización

Para la parte de visualización se utilizó la librería Highcharts desde la cual permitió mostrar graficamente los resultados obtenidos en las métricas.
Esta librería resultó muy útil gracias a su sencillez y su amplia cantidad de opciones para estadísticas, desde la típica gráfica de barras con histórico a complejas composiciones para detallar valores.
A su vez, para pasar los datos desde *hdfs* a *sql* se utilizó Sqoop que es una aplicación para la transferencia de datos entre bases de datos relacionales y Hadoop.
Se implementó es el siguiente script:

```
sqoop-export --table $1 --export-dir $2 --connect
jdbc:mysql://10.212.83.136/bigdata --username root --password root
--input-fields-terminated-by '\t'
```

El script toma como primer parámetro el nombre de la tabla y como segundo el path de la métrica en Hadoop. Por ejemplo, en el caso de worst_ads, la métrica se llama worst_ads porque los resultados se cargan en results/worst_ads, pero a la tabla se le puso worst_shows. entonces la invocación sería:

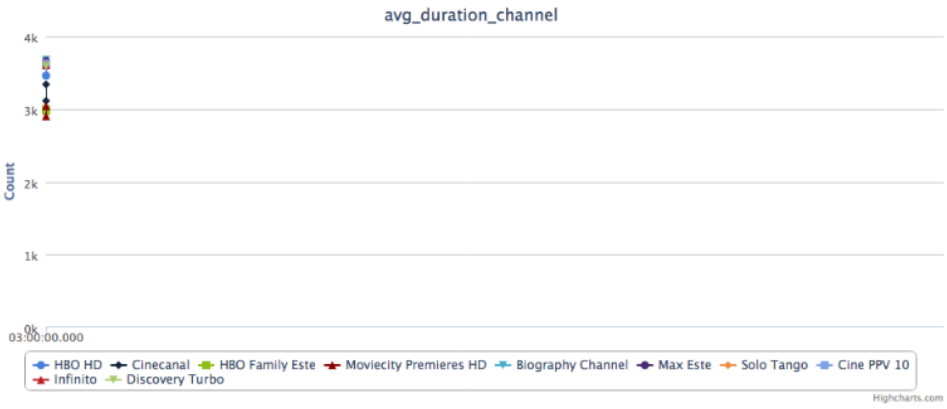
```
./exp.sh worst_shows results/worst_ads
```

Gráficos

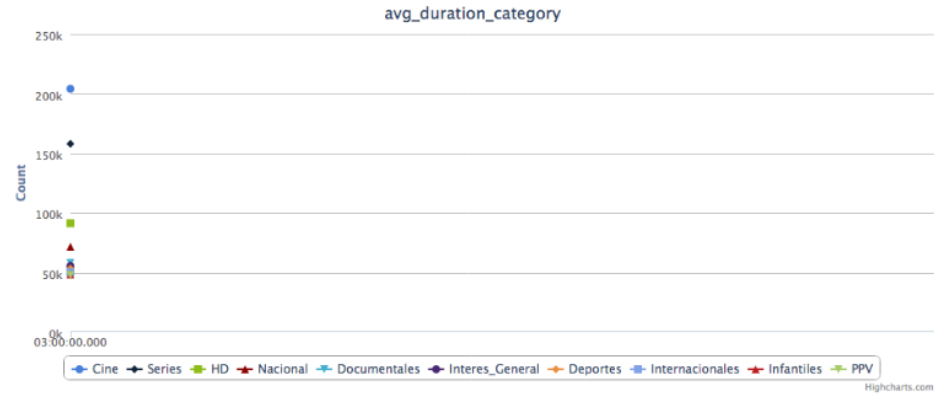
Se optó por diferentes tipos de gráficos dependiendo de la métrica, de modo de hacerlo lo más fácil posible la visualización de los datos.

Para los gráficos de línea que tienen demasiados categorías se utilizaron los diez resultados con más vistas. Para ello, se levantan todas las categorías, y se suman todos los valores para el intervalo de tiempo. Con ese valor obtenido se ordenan las categorías de mayor a menor y se trunca la lista con los primeros diez valores.

Métricas de tipo *Batch*:

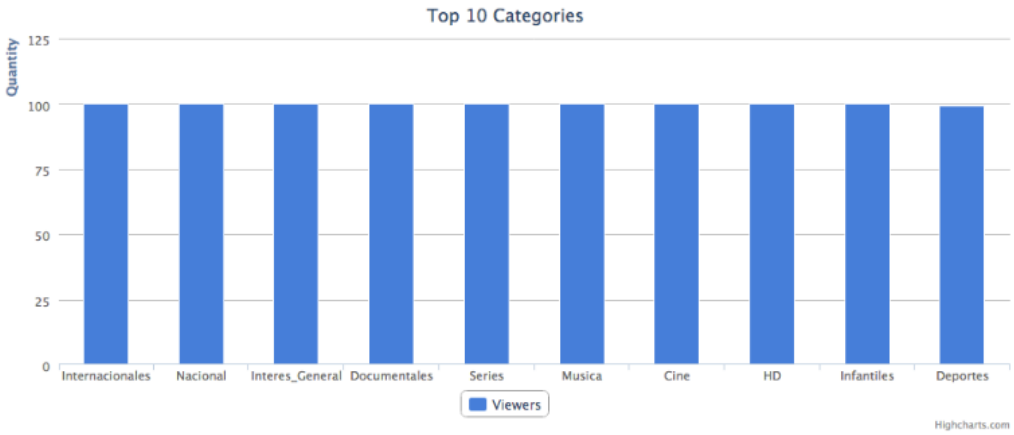


1 - Promedio de tiempo dedicado por televidentes a cada canal por día

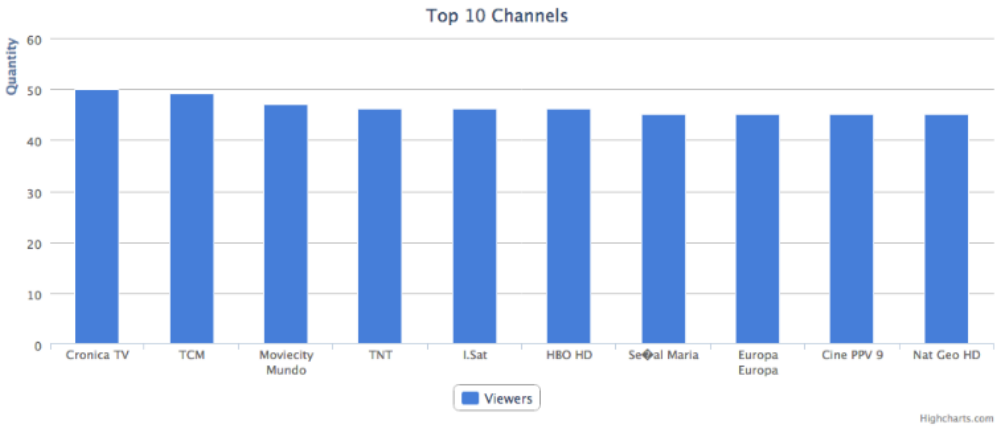


2 - Promedio de tiempo dedicado por televidentes a cada categoría por día

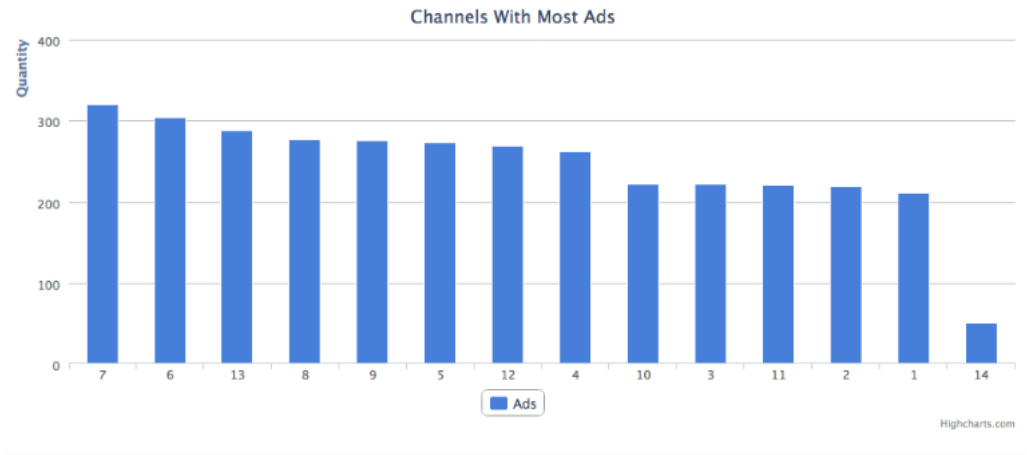
Las imágenes 1 y 2 muestran las métricas *"Promedio de tiempo dedicado por televidentes a cada canal y categoría por día"* calculadas con gráficos de líneas con un solo día. No se tomó en cuenta la opción de cambiar el tipo de gráficos según la cantidad de días ya que la base de datos se supone constante y grande.



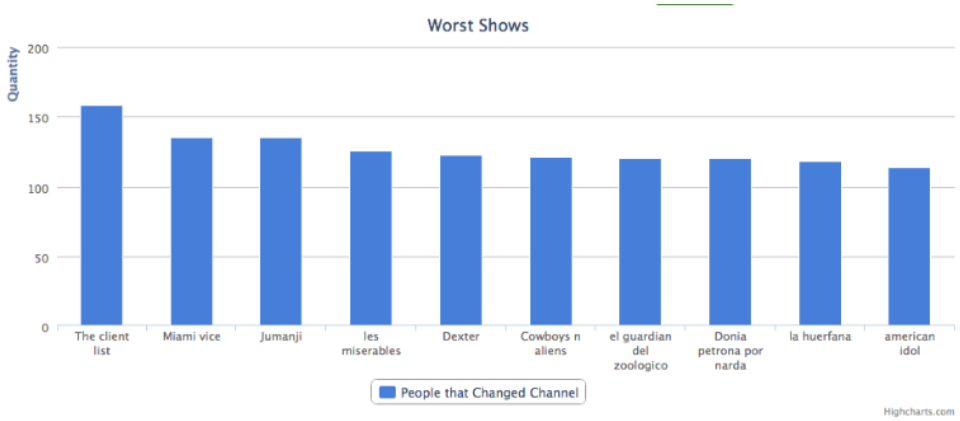
3 - Top 10 de categorias



4 - Top 10 de canales

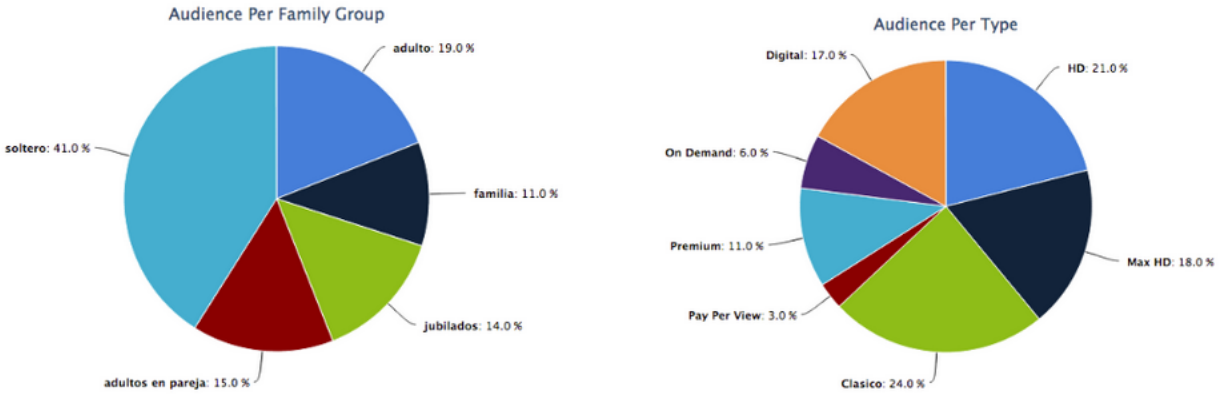


5 - Canales con más anuncios



6 - Los peores shows

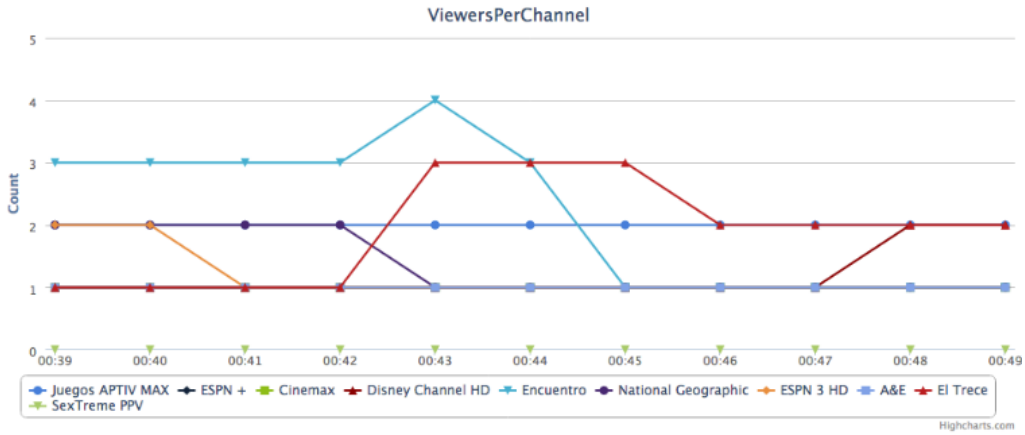
A las métricas correspondientes a las imágenes 3, 4, 5 y 6 se optó por mostrarlas en gráficos de barra ya que en esos casos se calcularon frecuencias. Los mismos resultaron muy útiles para visualizar de la manera más fácil esas métricas, ya que al poner las barras ordenadas de mayor a menor, permiten comparar los distintos valores calculados. A su vez, se graficaron las mismas con el mismo color, para evitar distintas percepciones por diferentes colores y mostrar los datos de la forma más objetiva posible.



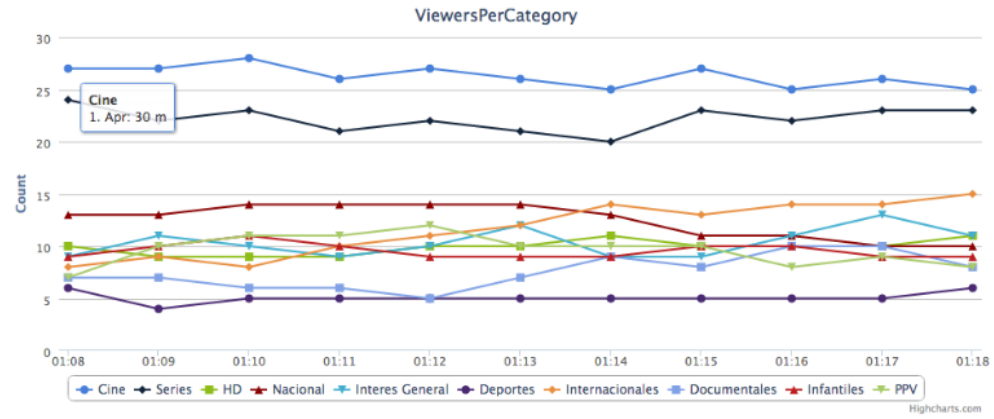
7 - Audiencia por grupo familiar y por tipo de cliente

Para las métricas de audiencia por grupo familiar y por tipo de cliente, se optaron gráficos de torta ya que es lo más útil para mostrar porcentajes de la distribución de una misma variable. En este caso, cómo se divide la audiencia para cada grupo familiar o para cada tipo de cliente.

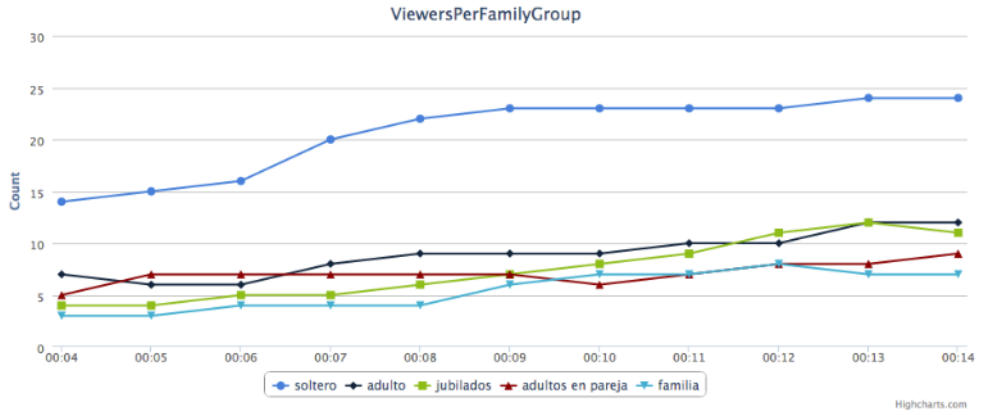
Métricas de tipo *Tiempo Real*:



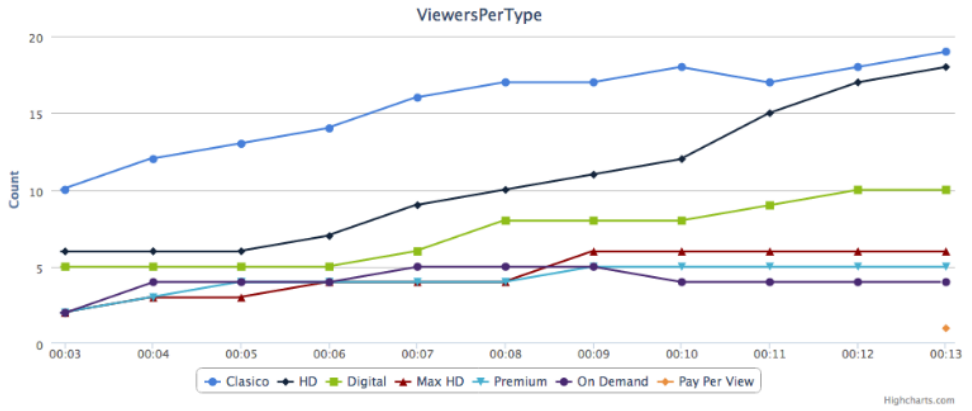
8 - Televidentes por canal



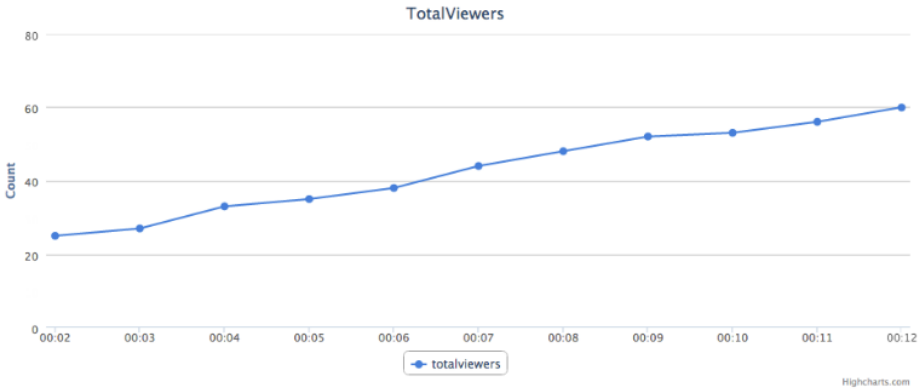
9 - Televidentes por categoria



10 - Televidentes por grupo familiar



11 - Televidentes por tipo de cliente



12 - Cantidad total de televidentes

Las imágenes 8, 9, 10, 11 y 12, muestran los gráficos correspondientes a las métricas de tipo Tiempo Real. Los mismos son del tipo gráficos de línea. Se consideró que es la forma más adecuada de mostrar la información, ya que permite visualizar cómo cambia cierto valor a través del tiempo.

Problemas encontrados en el desarrollo

Para métricas de tipo Batch

Los primeros problemas encontrados a la hora de implementar el trabajo práctico fueron de configuración. En otras palabras, errores producidos por pig debido a variables de entorno mal configuradas o faltantes, sobre todo a la hora de integrar Pig y HBase. Para empezar, si no se configura nada, Pig advertirá que no encuentra clases de Hbase (ClassNotFound). La solución es agregar el jar de HBase a la variable de entorno PIG_CLASSPATH. Sin embargo esto no alcanza; Pig tampoco encontrará clases de Zookeeper a la hora de ejecutar el script. Nuevamente la solución es incluir a esta variable todos los jars en el directorio *hbase/lib* , que tiene entre otras cosas el jar de zookeeper. Para hacer esto, se debe declarar:

```
export HBASE_HOME=/home/hadoop/hbase-0.94.6.1
export PIG_CLASSPATH=”`${HBASE_HOME}/bin/hbase classpath`:$PIG_CLASSPATH”
```

Incluso habiendo hecho esto, se encontraron más problemas. A la hora de leer datos de HBase, Zookeeper intentaba conectarse a 127.0.0.1 puerto 2281, y la conexión se rechazaba. Debido a que Zookeeper trabaja con nombres de hosts y no IPs, se buscó en /etc/hosts para verificar si existía una entrada mal hecha que dirija a 127.0.0.1, pero no se halló nada. Investigando sobre Hbase y zookeeper, se halló que esta dirección es cargada de una variable del *hbase-site.conf*, *hbase.zookeeper.quorum*. Tras verificar que la variable estaba correctamente seteada a un host correcto del cluster, se llegó a la conclusión de que esta configuración no estaba siendo cargada correctamente, y por lo tanto usaba el default que seguramente era *localhost*. Por lo tanto, se agregó:

```
export HBASE_CONF_DIR=/home/hadoop/hbase-0.94.6.1/conf
```

Por último, al correr un MapReduce, se hallaba un error de ClassNotFound de la clase Message de protobuf, a pesar de que el jar se encuentra en hbase/lib. Por lo tanto,
se registró manualmente el jar (con la instrucción register) en todo script que requiera acceso a HBase.

Campos opcionales de Json

A la hora de cargar datos de un archivo con un json, se verificó que la librería estándar de Pig para hacer esto no alcanzaba por sí sola, dado que no maneja correctamente campos opcionales (asume el mismo schema para toda entrada). Se encontraron varias soluciones para resolver esto, entre ellas:
1) Utilizar ElephantBird, una librería externa que incluye un loader de Json que deduce el esquema de los datos por sí sola.
2) Implementar un Loader de Json propio.
3) Implementar una UDF propio que devuelva los campos dado un string de Json, y usar un TextLoader.

La opción 1) se descartó dado que implicaba incluir una dependencia grande para una funcionalidad sencilla (deducir el schema no es nada sencillo; lo que se precisaba era sólo poder parsear json con campos opcionales y asignarles algún valor default si no se encuentra, y esto último sí se consideró sencillo).
La opción 2) se terminó descartando debido a que conlleva una complejidad mucho mayor a la 3, dado que el Loader reciba "chunks" que pueden estar desalineados con las líneas de json. El problema no es imposible de resolver, pero se prefirió utilizar la opción 3) por su simpleza, con la desventaja de ser menos eficiente (dado que primero se debe hacer el Load y luego parsear los campos del json).
Debido a esto, se implementaron dos UDFs. Una UDF, JsonSplitter, toma una línea de json y devuelve los campos que se precisaban, es decir, los campos están definidos en el código fuente y por lo tanto en el ejecutable, por lo que no es muy extensible. Los campos son devueltos en una tupla. El problema de esto es que si se invoca así:

```
json = FOREACH loadedJson GENERATE path.to.udf.JsonSplitter(line);
```

La UDF devuelve una tupla, y el GENERATE agrupa todos los valores a su derecha en una tupla. Por lo que el resultado es una tupla en una tupla. Sin embargo esto se puede solventar con un FLATTEN.
La segunda UDF implementada, JsonFieldAccess, toma dos valores obligatorios y uno opcional: la línea de json y el nombre del campo, y devuelve el valor asociado al nombre. En caso de no existir dicho nombre, devolverá el tercer parámetro, que corresponde al valor default del campo. En caso de no estar presente, se lanza un error (se asume que el campo era obligatorio al no tener valor default, y no se halló en el json). Se decidió que la UDF devuelva siempre un chararray (es decir String en java), dado que el usuario que invoca la UDF puede siempre castear el chararray recibido a int o long en caso de ser necesario. La segunda UDF resulta un poco más extensible, pero su invocación es un poco más verbosa. Además, tiene también la utilidad de que se puede cargar sólo los campos que son necesarios.

Schema de UDFs

Al utilizar una UDF que extiende de EvalFunc<T> y cuyo tipo T es una tupla o bag, es recomendable declarar su schema. Si no se hace, pig debe deducirlo, y muchas veces no puede hacerse para estos tipos de datos. Por lo tanto, si se utiliza describe se verá algo similar a:
{NULL}
Esto hace que se pierdan los nombres de todos los campos, y no se pueda acceder por nombre. Sí se puede seguir accediendo utilizando su posición, con $0, $1, etc. Sin embargo hacerlo es poco legible, inmantenible, y falla sólo cuando se ejecuta el MR si se accede a algo inválido.
Por lo tanto, se debe declarar el schema. Interesantemente, si uno lo declara mal, pig no tiene manera de saberlo de entrada. Al ejecutar el script, se verán errores cuyas causas son difíciles de determinar, como por ejemplo, ExecutionException, o ClassCastException: cannot cast Tuple to Long, o similar. Resultó esencial, por lo tanto, declarar el schema y hacerlo correctamente al utilizar UDFs que retornen tuplas o bags.

Para métricas de tipo Tiempo Real

El problema principal fue intentar paralelizar la topología, especialmente los Metric Counters. A pesar de que el problema es paralelizable en box_ids, resultó muy difícil sincronizar la etapa de emisión de las tablas de frecuencia. Había varios problemas, por ejemplo, no recibir registros en un bolt en un minuto. Para solventar esto se intentó implementar un Bolt sincronizador que se encargaba de emitir a k bolts paralelizados una advertencia al comenzar cada nuevo minuto. Sin embargo esto traía varios problemas más, por ejemplo, desfasaje entre el procesamiento de estos bolts (podía haber uno que esté tres minutos atrasado comparado a otro) y condiciones de carrera (qué pasa si el mensaje del bolt sincronizador llega antes que un registro del minuto anterior?). En vista de estos problemas, se decidió que los Metric Counters no sean paralelizables. Para solventar esto, las llamadas bloqueantes que hacen estos bolts se cachean.

El segundo problema fue la métrica de contar televidentes por edad y género. Dado que esa información no está persistida en HBase, resultó imposible calcularla.

Extensibilidad

Se intentó de que el programa sea lo más extensible posible, y que el agregado de

nuevas métricas que cuenten televidentes sea trivial. Por ello, las implementaciones de los Metric Counters son iguales para distintas métricas (la misma clase). Lo mismo ocurre con los Metric Dumpers. Para implementar una nueva métrica, se requiere implementar sólo dos funciones:

characteristicFunction: Es una función que toma una tupla (emitida por el spout) y retorna la *característica* que se desea analizar. Por ejemplo, si se trata de la cantidad de televidentes para un canal, la característica es el canal, y por lo tanto la función debe retornar el canal. Sin embargo, puede haber más de una característica asociada, como es el caso de las categorías. Por lo tanto se crearon dos implementaciones: una que toma una función de tupla a String y otra a lista de String.

mappingFunction: Es una función que toma un String y devuelve un String. La idea es dar la posibilidad de crear registros “formatteados”, verbosos o más presentables. Por ejemplo, la clave de una métrica puede ser el Id de algo. Sin embargo el Id es poco explicativo. En ese caso se puede implementar una mappingFunction que dado el Id retorne el nombre o similar. Es absolutamente opcional.

En vista de esto, es fácil ver que si se desea una métrica que cuente televidentes con algún criterio, basta implementar estas dos funciones (de las cuales una es opcional) y crear una tabla con el schema ya explicado anteriormente.
Si la métrica es más compleja y no consiste únicamente en contar televidentes, no se puede utilizar el esquema explicado anteriormente.

Posibles mejoras o extensiones

En pig, la cantidad máquinas que corren el Map depende de la cantidad de bloques en el HDFs. Sin embargo, la cantidad de máquinas que corren el reduce puede ser modificada utilizando la cláusula PARALLEL. El default es 1.
Sería interesante probar de correr las consultas batch con distintos valores de PARALLEL y ver qué valor produce los mejores resultados.

En Storm Idealmente sería conveniente poder paralelizar los MetricCounters, agrupando por box_id. El problema en sí es paralelizable, el problema fue que no se consiguió implementar. Sin embargo esto traería costos: un compromiso entre pérdida de performance por sincronización y memoria. Si se desea mantener la información del minuto actual en memoria, los bolts que procesen ese minuto más rápido deberán esperar a los más lentos. Si se desea mantener la información de más de un minuto en memoria (se podría implementar un margen de k minutos almacenados, y solo forzar la sincronización cuando la diferencia de minutos entre el más rápido y lento excede k) se debe pagar el costo en memoria.