BIG DATA

Serving layer

INTRODUCTION

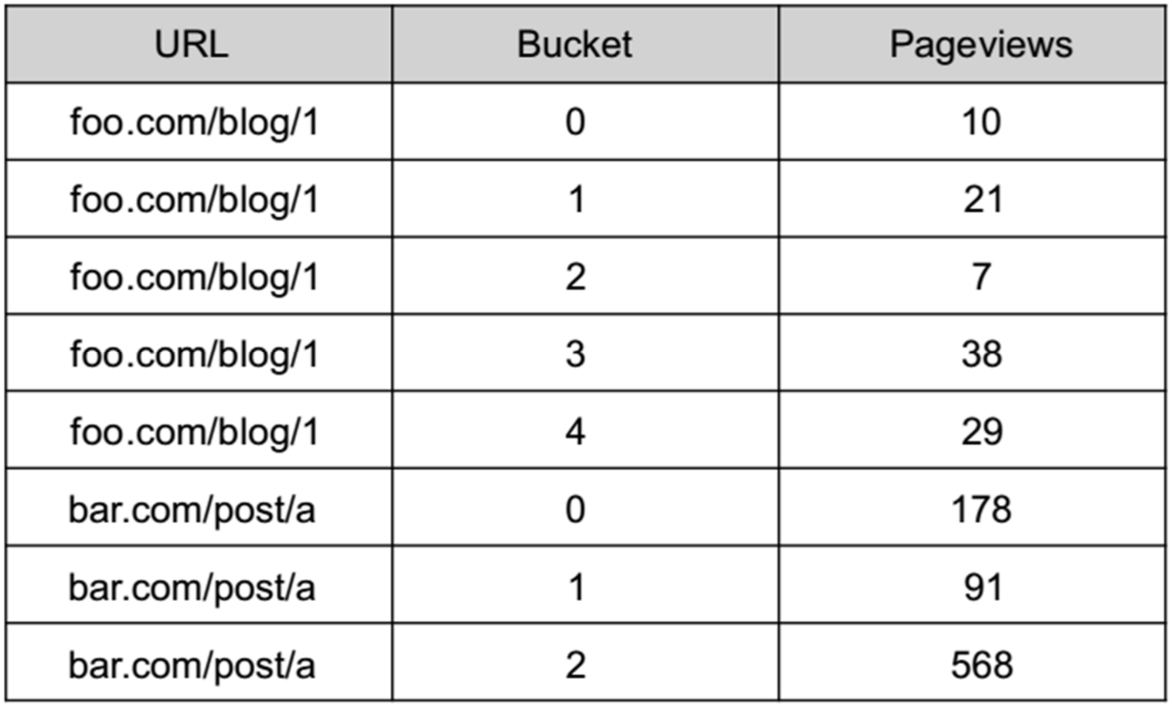
- In the last chapters you learned
  - how to precompute arbitrary views of any dataset by making use of batch computation
- In this chapter
  - you’ll learn how to access contents of views with low latency
  - We’ll present the full theory behind creating a simple, scalable, fault-tolerant, and general-purpose serving layer.
  - While investigating the serving layer, you’ll learn the following:
    - Indexing strategies to minimize latency, resource usage, and variance
    - The requirements for the serving layer in the Lambda Architecture
    - How the serving layer solves the long-debated normalization versus de-normalization problem

PERFORMANCE METRICS FOR THE SERVING LAYER

- As with the batch layer
  - the serving layer is distributed among many machines for scalability
  - The indexes of the serving layer are created, loaded, and served in a fully distributed manner
- Two main performance metrics for designing indexes:
  - Throughput: the number of queries that can be served within a given period of time
  - Latency: the time required to answer a single query

PERFORMANCE METRICS FOR THE SERVING LAYER

- Consider pageviews-over-time query:



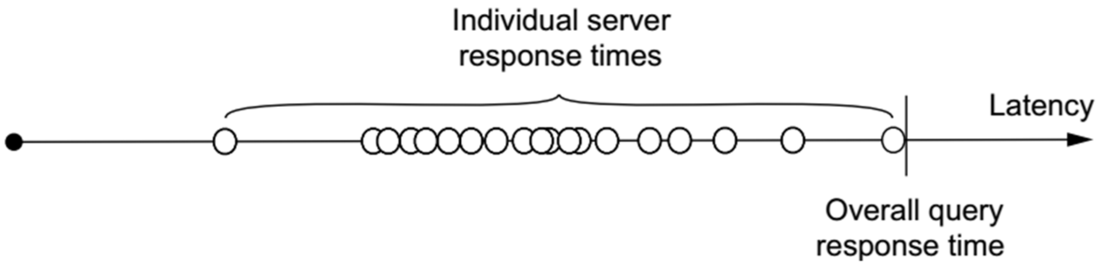
URL	Bucket	Pageviews
foo.com/blog/1	0	10
foo.com/blog/1	1	21
foo.com/blog/1	2	7
foo.com/blog/1	3	38
foo.com/blog/1	4	29
bar.com/post/a	0	178
bar.com/post/a	1	91
bar.com/post/a	2	568

PERFORMANCE METRICS FOR THE SERVING LAYER

- Consider pageviews-over-time query:
  - A straightforward way to index
    - use a key/value strategy with [URL, hour] pairs as keys and pageviews as values.
    - Partition index using the key
      - Pageview counts for the same URL would reside on different partitions.
      - Different partitions would exist on separate servers
      - retrieving a range of hours for a single URL involves fetching values from multiple servers

PERFORMANCE METRICS FOR THE SERVING LAYER

- Consider pageviews-over-time query:
  - A straightforward way to index
    - works but, has serious issues
      - Latency would be consistently high
        - Need to query numerous servers to get the pageview counts for a large range of hours
        - Response times of (even homogeneous) servers vary
          - Different loads
          - Garbage collection
        - Overall query response time is limited by the speed of the slowest server
        - The more servers a query touches, the higher the overall latency of the query
          - more servers => more likelihood that at least one will respond slowly
          - worst-case performance of one server => common-case performance of queries

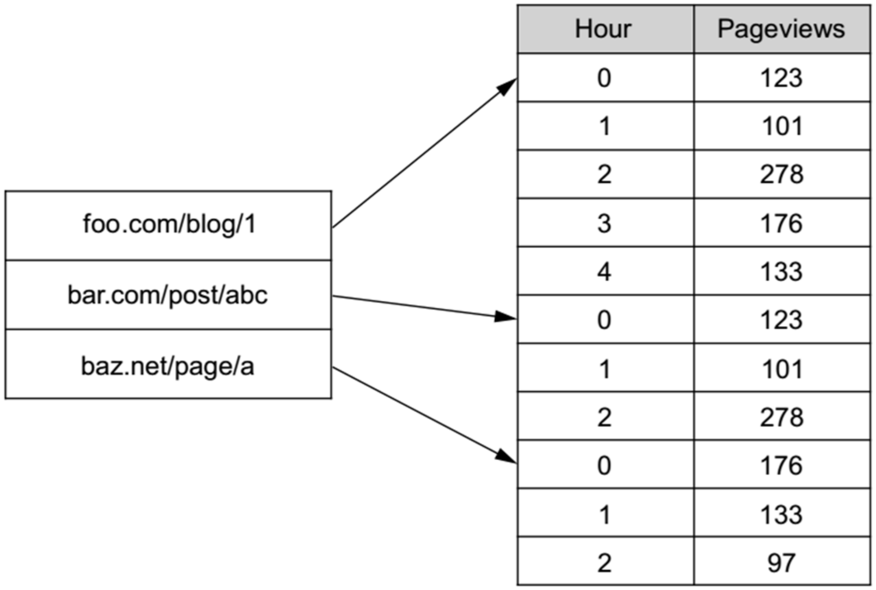


PERFORMANCE METRICS FOR THE SERVING LAYER

- Consider pageviews-over-time query:
  - A straightforward way to index
    - works but, has serious issues
      - Poor throughput
        - Retrieving a value for a single key requires a disk seek
        - A single query may fetch values for dozens or more keys.
        - Disk seeks are expensive operations
          - finite number of disks in cluster => hard limit to the number of disk seeks per second
        - Suppose that on average:
          - a query fetches 20 keys per query
          - the cluster has 100 disks
          - each disk can perform 500 seeks per second.
        - In this case, the cluster can only serve 2,500 queries per second

PERFORMANCE METRICS FOR THE SERVING LAYER

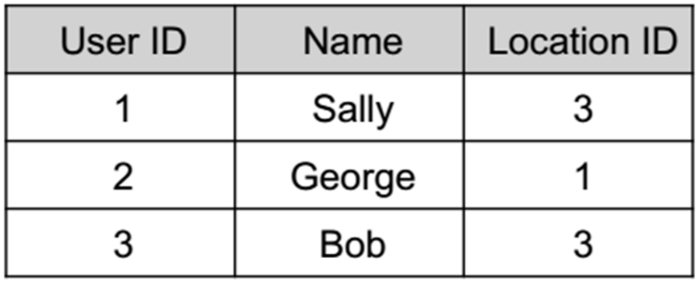
- Consider pageviews-over-time query:
    - Another indexing strategy
        - store the pageviews information for a single URL on the same partition sequentially
        - Fetching the pageviews only require a single seek and scan
            - Scans are extremely cheap relative to seeks
            - more resource efficiency.
        - Only a single server needs to be contacted per query
            - no longer subject to the issues of the previous strategy



Hour	Pageviews
0	123
1	101
2	278
3	176
4	133
0	123
1	101
2	278
0	176
1	133
2	97

SOLUTION TO THE NORMALIZATION/ DENORMALIZATION PROBLEM

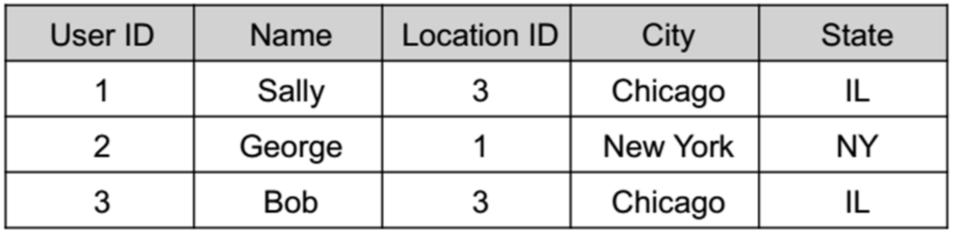
- Normalized



User ID	Name	Location ID
1	Sally	3
2	George	1
3	Bob	3

Location ID	City	State	Population
1	New York	NY	8.2M
2	San Diego	CA	1.3M
3	Chicago	IL	2.7M

- De-normalized



User ID	Name	Location ID	City	State
1	Sally	3	Chicago	IL
2	George	1	New York	NY
3	Bob	3	Chicago	IL

Location ID	City	State	Population
1	New York	NY	8.2M
2	San Diego	CA	1.3M
3	Chicago	IL	2.7M

REQUIREMENTS FOR A SERVING LAYER DATABASE

- Four requirements:
  - Batch writable
  - Scalable
  - Random reads
  - Fault-tolerant
- Amazing property
  - Does not require random writes
    - responsible for the majority of the complexity
    - E.g.
      - need for compaction
      - need to synchronize reads and writes

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Pageviews over time
  - Recall: batch view computes the bucketed counts for hourly, daily, weekly, monthly, and yearly granularities
    - minimizes the total number of retrieved values to resolve a query
  - Having key-to-sorted-map index makes these higher granularities redundant
    - extremely cheap to read all sequentially stored values for a range all at once
    - Example:
      - 12 bytes for each entry (4 bytes for the bucket number and 8 bytes for the value)
      - approximately 17,500 values for a two-year period
      - Total amount of 205 KB must be retrieved

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Uniques over time
  - The only way with perfect accuracy: compute the unique count on the fly
    - Too expensive
  - Alternate: an approximation like the HyperLogLog algorithm
    - requires information on the order of 1 KB to estimate set cardinalities of up to one billion with a maximum 2% error rate

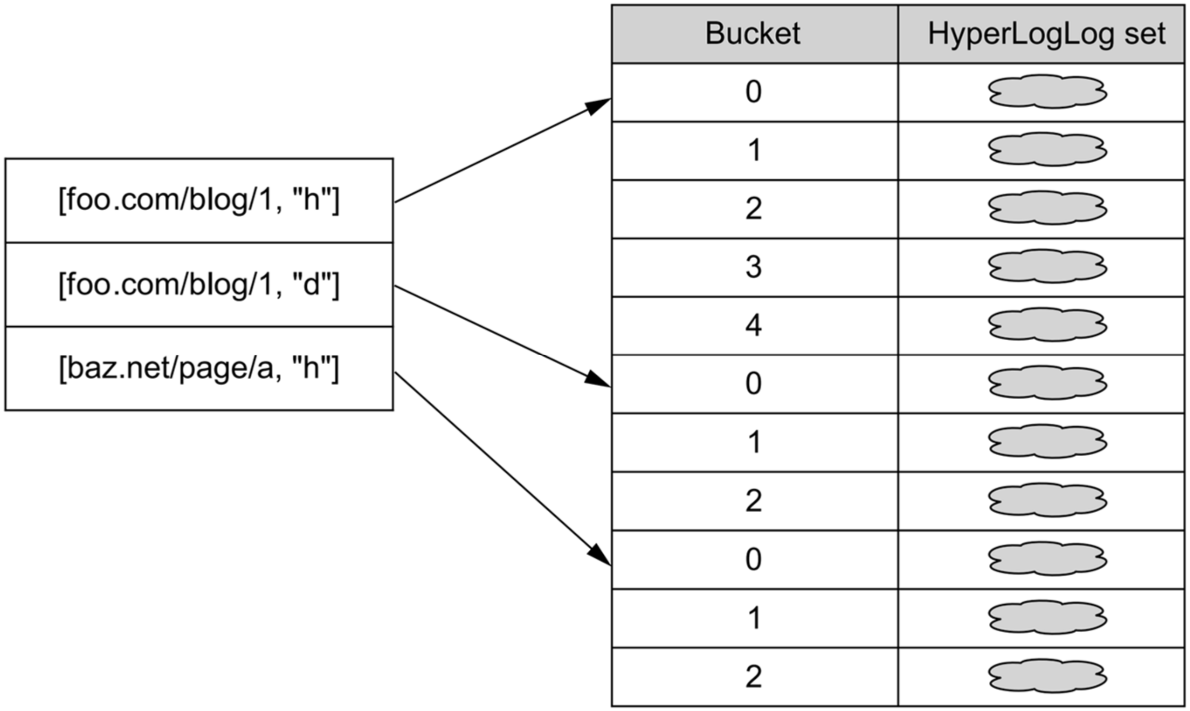
```
interface HyperLogLog {
  long size();
  void add(Object o);
  HyperLogLog merge(HyperLogLog... otherSets);
}
```

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Uniques over time
  - Very similar to pageviews over time but with big differences:
    - HyperLogLog sets used for buckets are significantly larger
    - More data to read
      - Having hourly granularity and 1024 bytes for HyperLogLog set size
      - 17 MB of HyperLogLog information for a two-year query
      - 60 ms just for reading the information with a hard disk with a read throughput of 300 MB/s
    - Merging HyperLogLog sets is expensive
    - making use of the higher granularities would be better

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Uniques over time
  - The key is a compound key of URL and granularity
  - The indexes are partitioned solely by the URL, not by both the URL and granularity

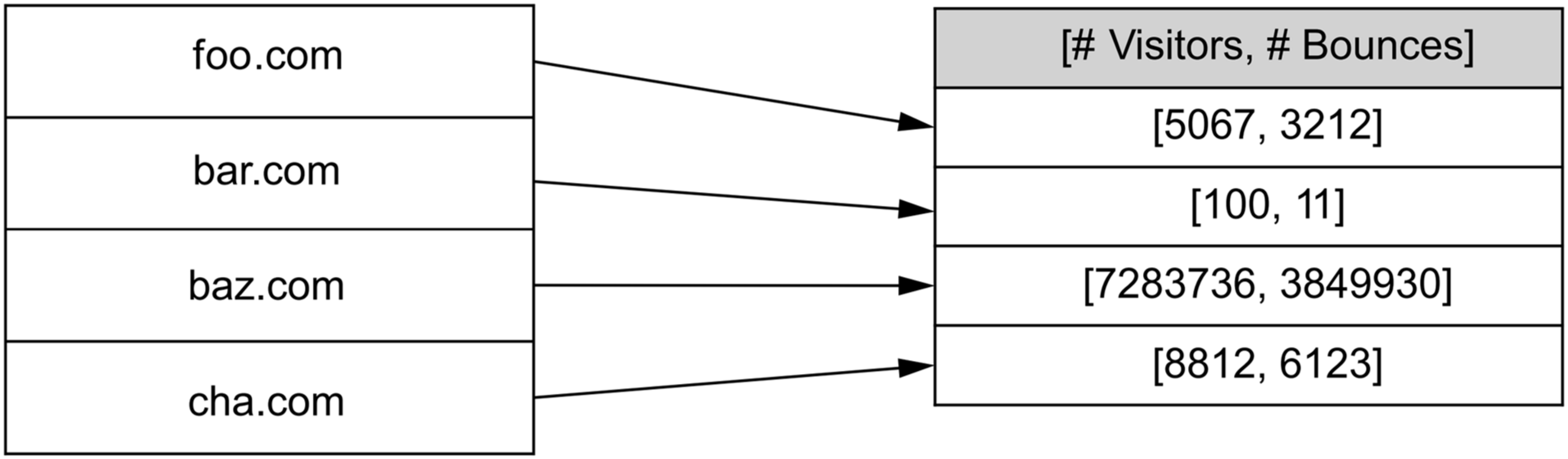


[foo.com/blog/1, "h"]
[foo.com/blog/1, "d"]
[baz.net/page/a, "h"]

Bucket	HyperLogLog set
0	☁
1	☁
2	☁
3	☁
4	☁
0	☁
1	☁
2	☁
0	☁
1	☁
2	☁

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Bounce-rate analysis
  - only requires a key/value index



foo.com

bar.com

baz.com

cha.com

[# Visitors, # Bounces]

[5067, 3212]

[100, 11]

[7283736, 3849930]

[8812, 6123]

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - First attempt: using a key-to-set database

```
interface KeyToSetDatabase {
   Set getSet(Object key);
   void addToSet(Object key, Object val);
}
```

  - Two pieces to any fully incremental approach:
    - the write side
    - the read side

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - First attempt: using a key-to-set database
    - The write side
      - key in the database: pair of [URL, hour bucket]
      - Value: the set of all UserIDs to visit that URL in that hour bucket
      - Whenever a new pageview is received
        - UserID is added to the appropriate bucket

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - First attempt: using a key-to-set database
    - The read side
      - Queries are resolved by
        - fetching all buckets in the range of the query
        - merging the sets together
        - computing the unique count of that set

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - First attempt: using a key-to-set database
    - Straightforward, but with a lot of problems
      - The database is very large space-wise
      - Very large number of database lookups for a query over a large range
        - a one-year period contains about 8,760 buckets.
      - For popular websites, even individual buckets could have tens of millions of elements

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - Second approach: using a key-to-HyperLogLog database
    - Key: pair of [URL, hour bucket]
    - Value: a HyperLogLog set representing all UserIDs that visit that URL in that hour.
    - Write side: simply adds the UserID to the appropriate bucket's HyperLogLog set
    - Read side
      - fetches all HyperLogLog sets in that range
      - merges them together
      - gets the count

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - Second approach: using a key-to-HyperLogLog database
    - Enormous space savings of HyperLogLog => everything is more efficient
      - Individual buckets are now guaranteed to be small
      - The database as a whole is significantly more space-efficient
    - But still has the problem of queries over large ranges
      - Fix: change the key to be a triplet of [URL, hour bucket, granularity]
    - Write side on a new pageview
      - Add UserID to the HyperLogLog set of the target bucket with appropriate granularity
    - Read side
      - The minimum number of buckets are read to compute the result

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - Second approach: using a key-to-HyperLogLog database
    - This is a very satisfactory approach to the problem
      - fast for all queries
      - space-efficient
      - easy to understand
      - straightforward to implement

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - But, what about equivs?
    - A terrible result: There’s no way to use HyperLogLog
      - A HyperLogLog set doesn’t know what elements are within it

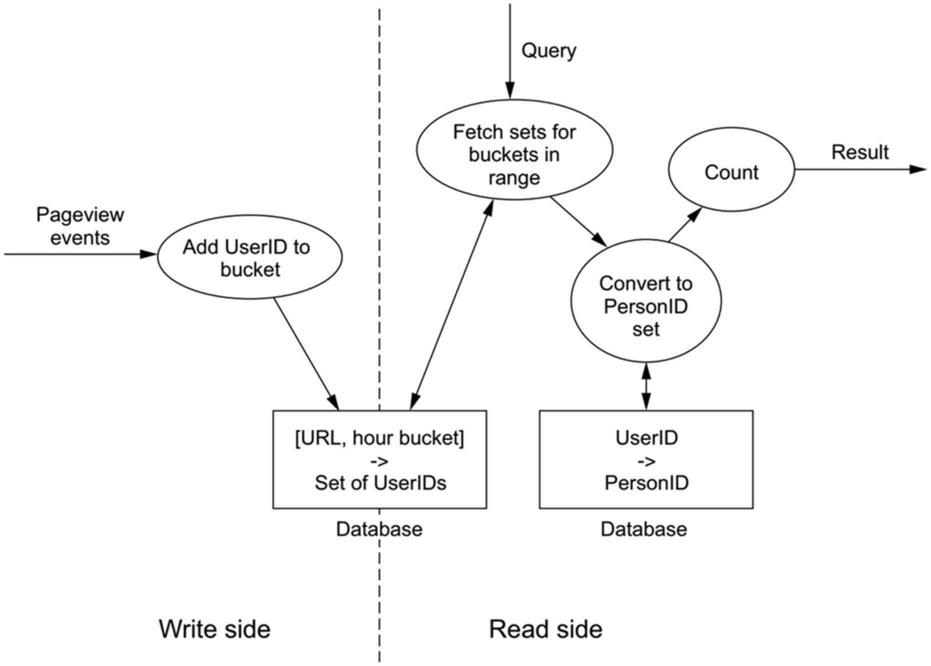
DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
    - But, what about equivs?
        - Lets go back to the first attempt
            - a set of UserIDs is stored for every [URL, hour bucket] pair
            - Suppose that: only store one UserID per person
                - You should iterate over the entire database on a new equiv
                - Or use a second index
                    - UserID -> set of all buckets the UserID exists in
                - What if a search engine bot visits every URL every hour?
                    - That UserID's bucket will contain all buckets in database
                    - Highly impractical

Key (URL, Hour bucket)	Set of UserIDs
"foo.com/page1", 0	A, B, C
"foo.com/page1", 1	A, D
"foo.com/page1", 2	A, C, F
"foo.com/page1", 102	A, B, C, G

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - But, what about equivs?
    - Lets go back to the first attempt
      - Another approach: handling equivs on the read side (first attempt)

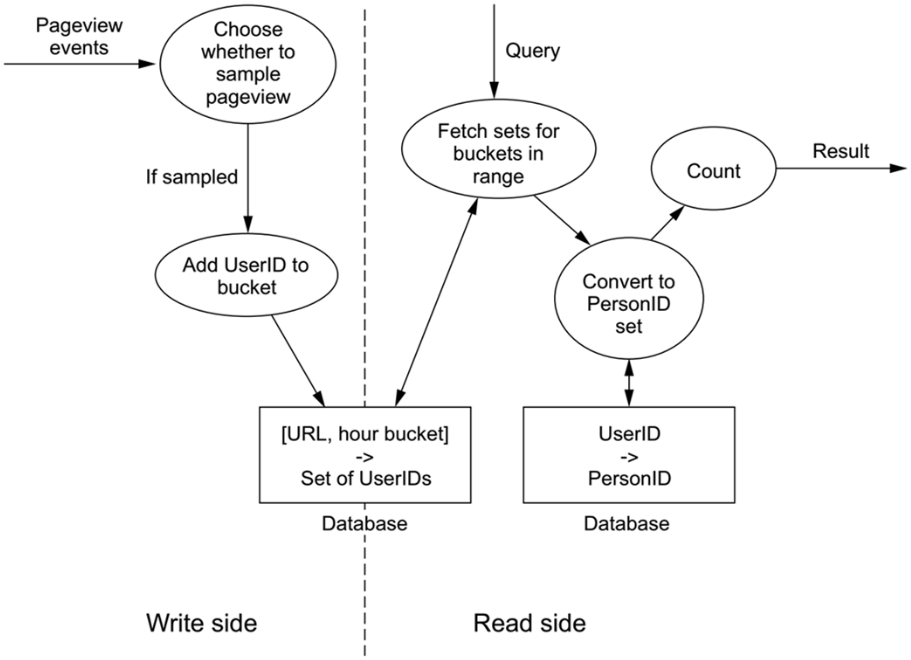


DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - But, what about equivs?
    - Lets go back to the first attempt
      - Another approach: handling equivs on the read side (first attempt)
        - it’s far too expensive
        - Imagine a query that has 100 million uniques.
          - you’d have to fetch many gigabytes of information to get the UserID set
          - then do 100 million lookups into the UserID-to-PersonID index.
          - There’s no way that work will ever complete in just a few milliseconds

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - But, what about equivs?
    - Lets go back to the first attempt
      - Another approach: handling equivs on the read side (second attempt)



DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Contrasting with a fully incremental solution
  - But, what about equivs?
    - Lets go back to the first attempt
      - Another approach: handling equivs on the read side (second attempt)
        - Good news: we finally have a viable approach that can be made performant.
        - Bad news: this comes with some caveats.
          - The level of accuracy is not nearly the same as HyperLogLog
          - Good throughput requires special hardware for the UserID-to-PersonID index
            - UserID sets need at least 100 elements for reasonable error rates
            - At least 100 lookups into UserID-to-PersonID index during queries
            - Each lookup requires at least one seek
            - You should use SSD
            - Or ensure that the UserID-to-PersonID index is kept completely in memory

DESIGNING A SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Comparing to the Lambda Architecture solution
  - Fully incremental solution is worse in every respect than the Lambda Architecture solution
    - It must use an approximation technique with significantly higher error rates
    - It has worse latency
    - It requires special hardware to achieve reasonable throughput
    - What makes all the difference?
      - A fully incremental solution has to handle equivs as they come in
      - The batch layer looks at all the data at once
        - The equivs are handled first
        - Then the views are created with that out of the way
        - You gain the ability to use a far more efficient strategy

ILLUSTRATION

- Basics of ElephantDB
  - It is a key/value database
    - both keys and values are stored as byte arrays.
  - It partitions the batch views over a fixed number of shards
  - Each server is responsible for some subset of those shards
  - Sharding scheme: The pluggable function that assigns keys to shards
  - Once assigned to a shard
    - The key/value is stored in a local indexing engine
    - BerkeleyDB is the default
    - But the engine is configurable
      - it could be any key/value indexing engine that runs on a single machine

ILLUSTRATION

- Basics of ElephantDB
  - Two aspects to ElephantDB
    - View creation
      - occurs in a MapReduce job at the end of the batch layer workflow
      - the generated partitions are stored in the distributed filesystem
    - View serving.
      - a dedicated ElephantDB cluster loads the shards from the distributed filesystem
      - interacts with clients that support random read requests.

ILLUSTRATION

- Basics of ElephantDB
  - View creation in ElephantDB
    - Shards are created by a MapReduce job
      - Input is a set of key/value pairs.
      - The number of reducers is configured to be the number of ElephantDB shards
      - The keys are partitioned to the reducers using the specified sharding scheme
      - Each reducer is responsible for producing exactly one shard
      - Each shard is then indexed (BerkeleyDB) and uploaded to the distributed filesystem

ILLUSTRATION

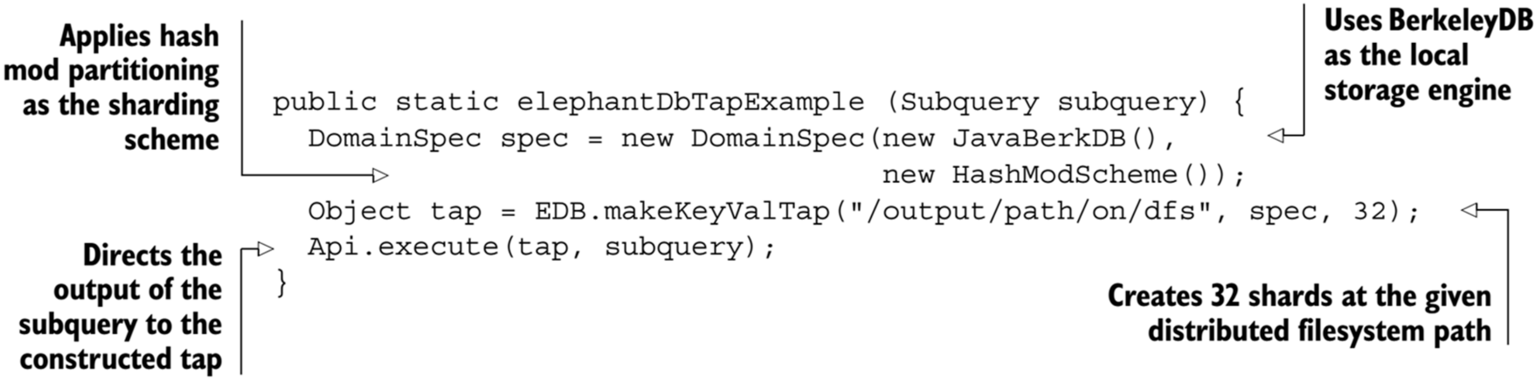
- Basics of ElephantDB
  - View serving in ElephantDB
    - cluster is composed of a number of machines that divide the work of serving the shards.
    - shards are evenly distributed among the servers.
    - also supports replication
      - each shard is redundantly hosted across a predetermined number of servers
    - When a server detects that a new version of a shard is available
      - it does a throttled download of the new partition
      - Upon completing the download, it switches to the new partition and deletes the old one
      - Then the contents of the batch views are accessible via a basic API

ILLUSTRATION

- Basics of ElephantDB
    - Using ElephantDB
        - It is straightforward to use.
        - There are three separate aspects
            - creating shards
            - setting up a cluster to serve requests
            - using the client API to query the batch views

ILLUSTRATION

- Basics of ElephantDB
    - Creating Elephantdb Shards
        - It provides a tap to automate the shard creation process
            - tap abstraction makes it simple to create a set of shards
        - Having a subquery that generates key/value pairs
            - creating the ElephantDB view is as simple as executing that subquery into the tap



Applies hash mod partitioning as the sharding scheme

Uses BerkeleyDB as the local storage engine

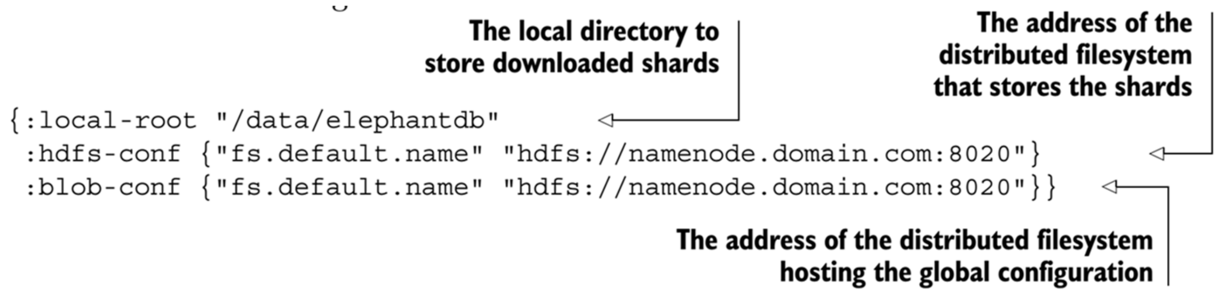
```
public static elephantDbTapExample (Subquery subquery) {
  DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                   new HashModScheme());
  Object tap = EDB.makeKeyValTap("/output/path/on/dfs", spec, 32);
  Api.execute(tap, subquery);
}
```

Directs the output of the subquery to the constructed tap

Creates 32 shards at the given distributed filesystem path

ILLUSTRATION

- Basics of ElephantDB
  - Setting Up An Elephantdb Cluster
    - Two required configurations
      - local configuration



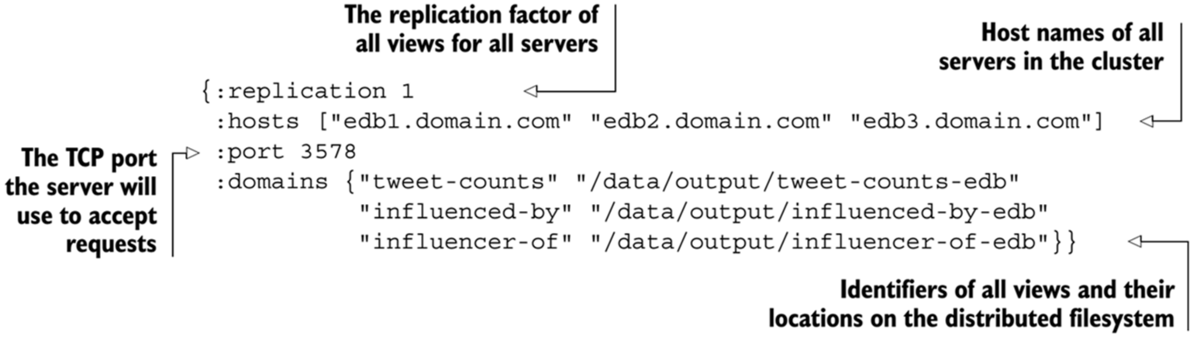
The local directory to store downloaded shards

The address of the distributed filesystem that stores the shards

```
{:local-root "/data/elephantdb"
 :hdfs-conf {"fs.default.name" "hdfs://namenode.domain.com:8020"}
 :blob-conf {"fs.default.name" "hdfs://namenode.domain.com:8020"}}
```

The address of the distributed filesystem hosting the global configuration

      - global configuration



The replication factor of all views for all servers

Host names of all servers in the cluster

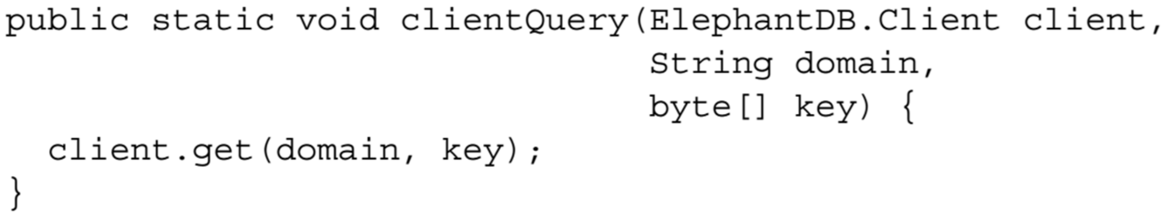
The TCP port the server will use to accept requests

```
{:replication 1
 :hosts ["edb1.domain.com" "edb2.domain.com" "edb3.domain.com"]
 :port 3578
 :domains {"tweet-counts" "/data/output/tweet-counts-edb"
           "influenced-by" "/data/output/influenced-by-edb"
           "influencer-of" "/data/output/influencer-of-edb"}}
```

Identifiers of all views and their locations on the distributed filesystem

ILLUSTRATION

- Basics of ElephantDB
  - Querying An Elephantdb Cluster
    - simple API for issuing queries
    - After connecting to any ElephantDB server, you can issue queries



```
public static void clientQuery(ElephantDB.Client client,
                               String domain,
                               byte[] key) {
  client.get(domain, key);
}
```

    - If the connected server doesn’t store the requested key locally, it will communicate with the other servers in the cluster to retrieve the desired values.

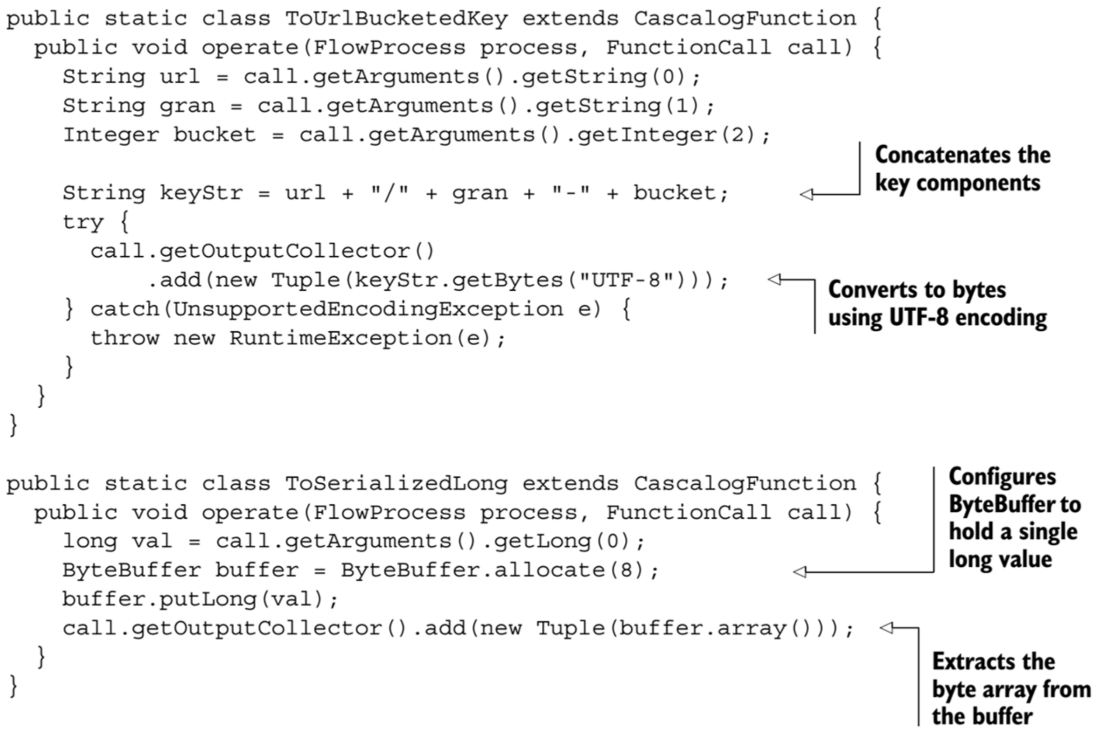
BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Pageviews over time
  - Ideal view: an index from key to sorted map
  - ElephantDB only supports key/value indexing
  - The ideal view is not possible with ElephantDB
  - All the granularities should be indexed into the view

URL	Granularity	Bucket	Pageviews
foo.com/blog/1	h	0	10
foo.com/blog/1	h	1	21
foo.com/blog/1	h	2	7
foo.com/blog/1	w	0	38
foo.com/blog/1	m	0	38
bar.com/post/a	h	0	213
bar.com/post/a	h	1	178
bar.com/post/a	h	2	568

BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Pageviews over time
  - serializations for composite keys and the pageview values



```
public static class ToUrlBucketedKey extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String url = call.getArguments().getString(0);
    String gran = call.getArguments().getString(1);
    Integer bucket = call.getArguments().getInteger(2);

    String keyStr = url + "/" + gran + "-" + bucket;
    try {
      call.getOutputCollector()
          .add(new Tuple(keyStr.getBytes("UTF-8")));
    } catch(UnsupportedEncodingException e) {
      throw new RuntimeException(e);
    }
  }
}

public static class ToSerializedLong extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    long val = call.getArguments().getLong(0);
    ByteBuffer buffer = ByteBuffer.allocate(8);
    buffer.putLong(val);
    call.getOutputCollector().add(new Tuple(buffer.array()));
  }
}
```

Concatenates the key components

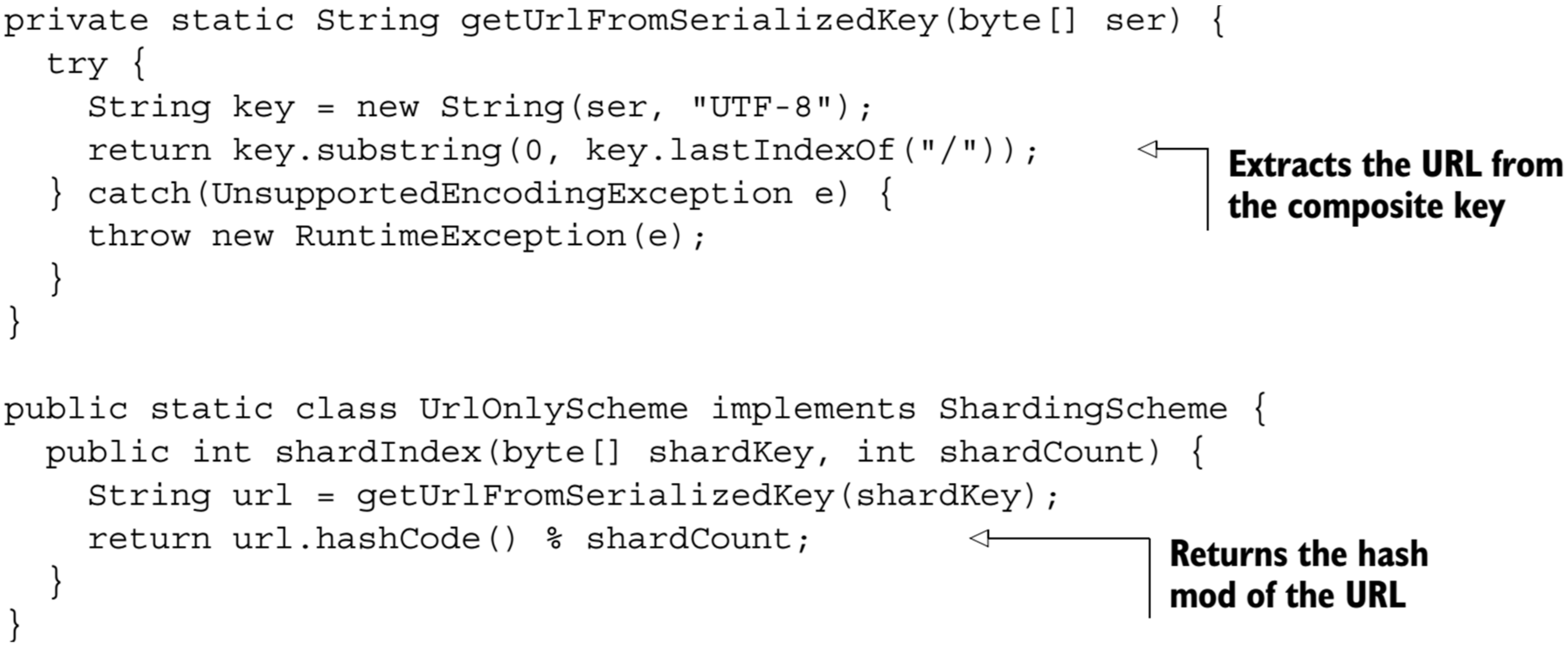
Converts to bytes using UTF-8 encoding

Configures ByteBuffer to hold a single long value

Extracts the byte array from the buffer

BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Pageviews over time
  - avoid the variance problem with a custom ShardingScheme



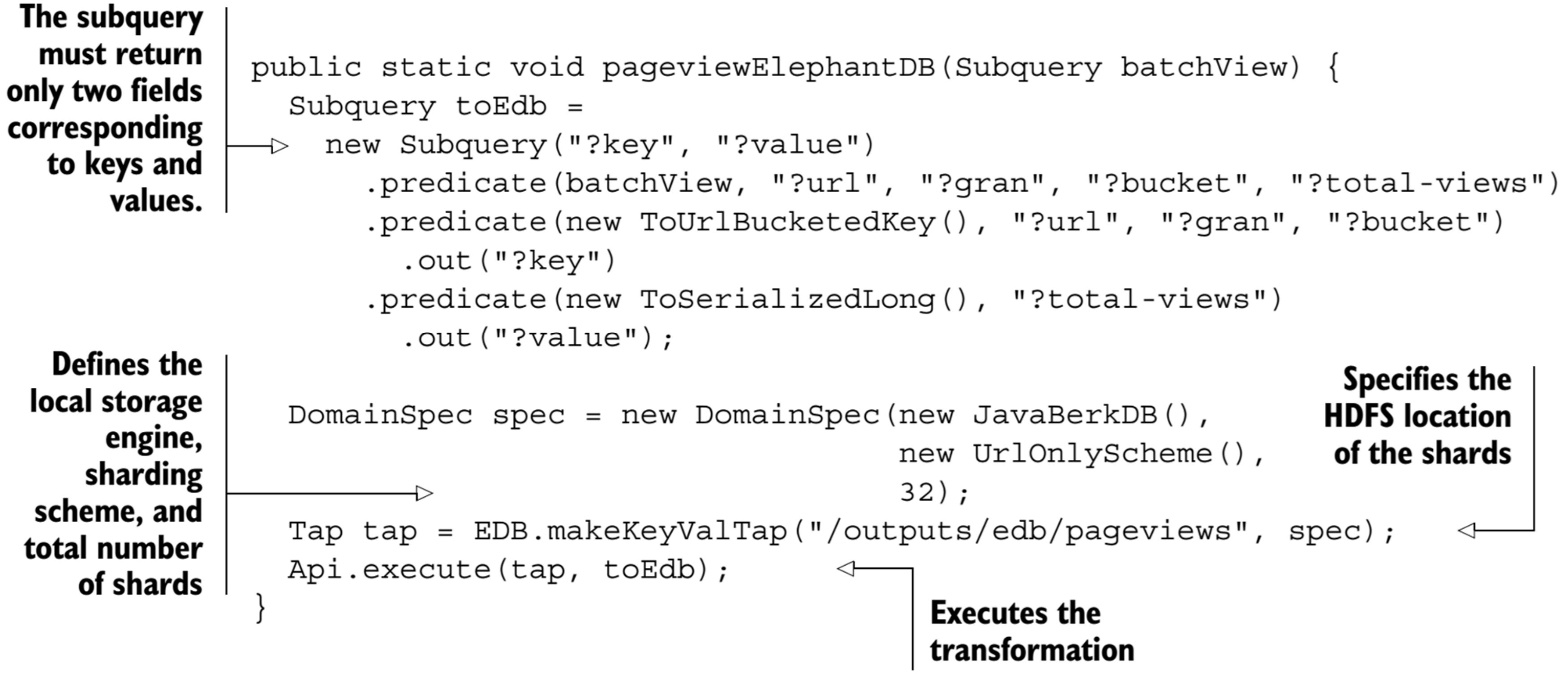
```
private static String getUrlFromSerializedKey(byte[] ser) {
  try {
    String key = new String(ser, "UTF-8");
    return key.substring(0, key.lastIndexOf("/"));
  } catch(UnsupportedEncodingException e) {
    throw new RuntimeException(e);
  }
}

public static class UrlOnlyScheme implements ShardingScheme {
  public int shardIndex(byte[] shardKey, int shardCount) {
    String url = getUrlFromSerializedKey(shardKey);
    return url.hashCode() % shardCount;
  }
}
```

Extracts the URL from the composite key

Returns the hash mod of the URL

BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Pageviews over time
  - create the ElephantDB tap and put the pieces together



```
public static void pageviewElephantDB(Subquery batchView) {
  Subquery toEdb =
    new Subquery("?key", "?value")
      .predicate(batchView, "?url", "?gran", "?bucket", "?total-views")
      .predicate(new ToUrlBucketedKey(), "?url", "?gran", "?bucket")
        .out("?key")
      .predicate(new ToSerializedLong(), "?total-views")
        .out("?value");

  DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                   new UrlOnlyScheme(),
                                   32);
  Tap tap = EDB.makeKeyValTap("/outputs/edb/pageviews", spec);
  Api.execute(tap, toEdb);
}
```

The subquery must return only two fields corresponding to keys and values.

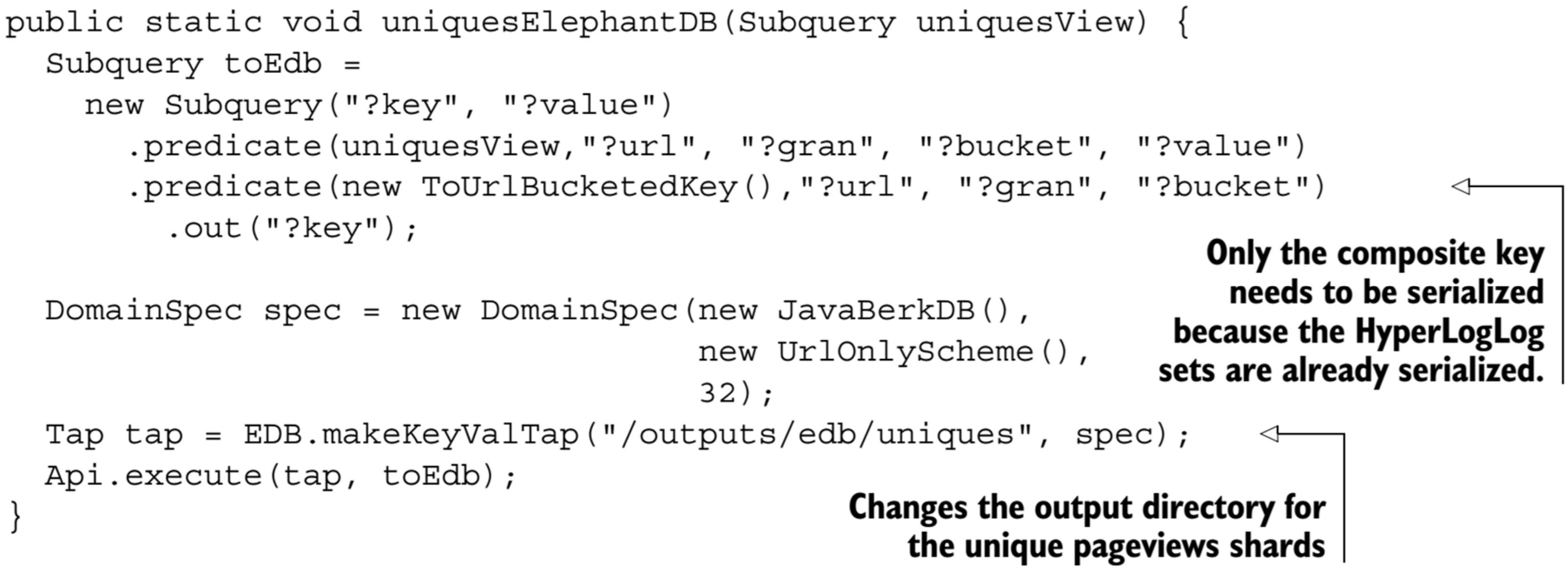
Defines the local storage engine, sharding scheme, and total number of shards

Specifies the HDFS location of the shards

Executes the transformation

BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Uniques over time
  - Ideal index cannot be implemented
  - Strategy: similar to the one used by pageviews over time
    - The only difference is that uniques over time stores HyperLogLog sets



```
public static void uniquesElephantDB(Subquery uniquesView) {
  Subquery toEdb =
    new Subquery("?key", "?value")
      .predicate(uniquesView,"?url", "?gran", "?bucket", "?value")
      .predicate(new ToUrlBucketedKey(),"?url", "?gran", "?bucket")
        .out("?key");

  DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                   new UrlOnlyScheme(),
                                   32);
  Tap tap = EDB.makeKeyValTap("/outputs/edb/uniques", spec);
  Api.execute(tap, toEdb);
}
```

Only the composite key needs to be serialized because the HyperLogLog sets are already serialized.

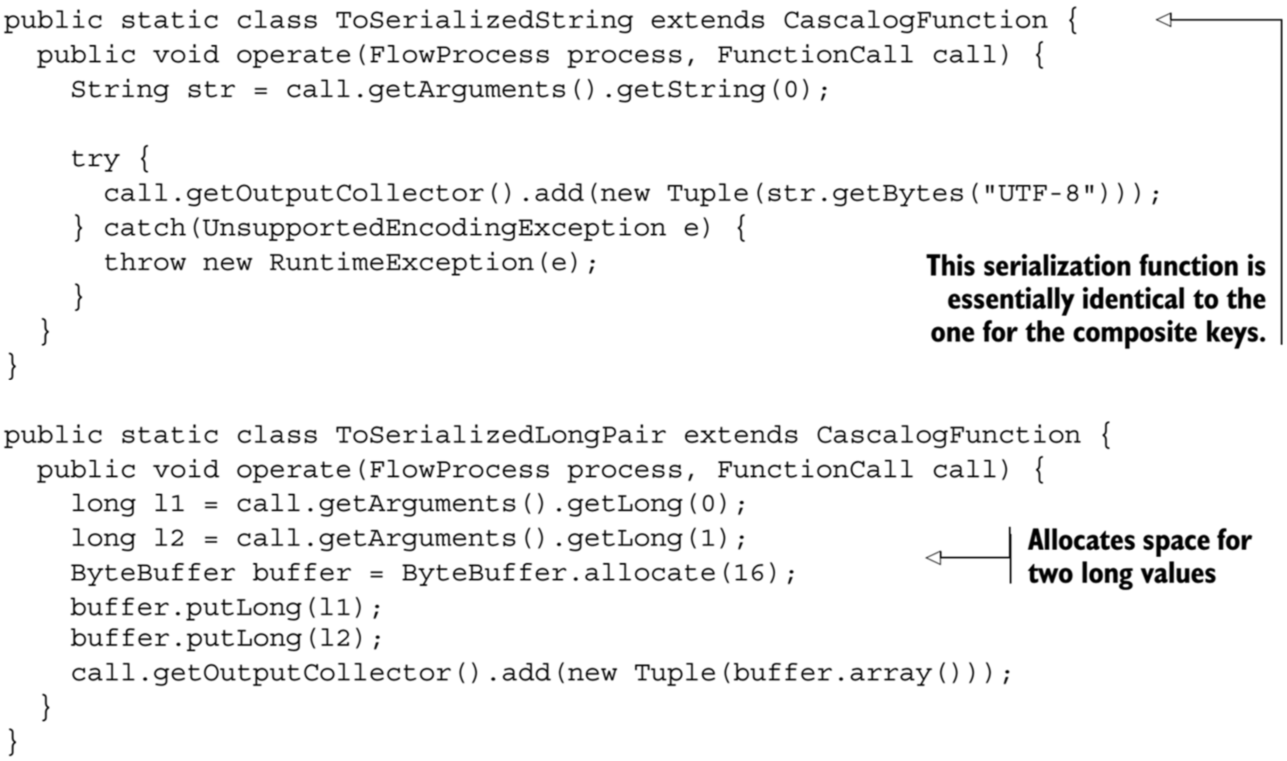
Changes the output directory for the unique pageviews shards

BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Uniques over time
  - Ideal serving layer database would know how to handle HyperLogLog sets natively
    - Complete queries on the server.
    - server should merge the sets and return only the cardinality
    - This would maximize efficiency
      - avoiding the network transfer of any HyperLogLog sets during queries

BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Bounce-rate analysis
  - ideal view is a key/value index



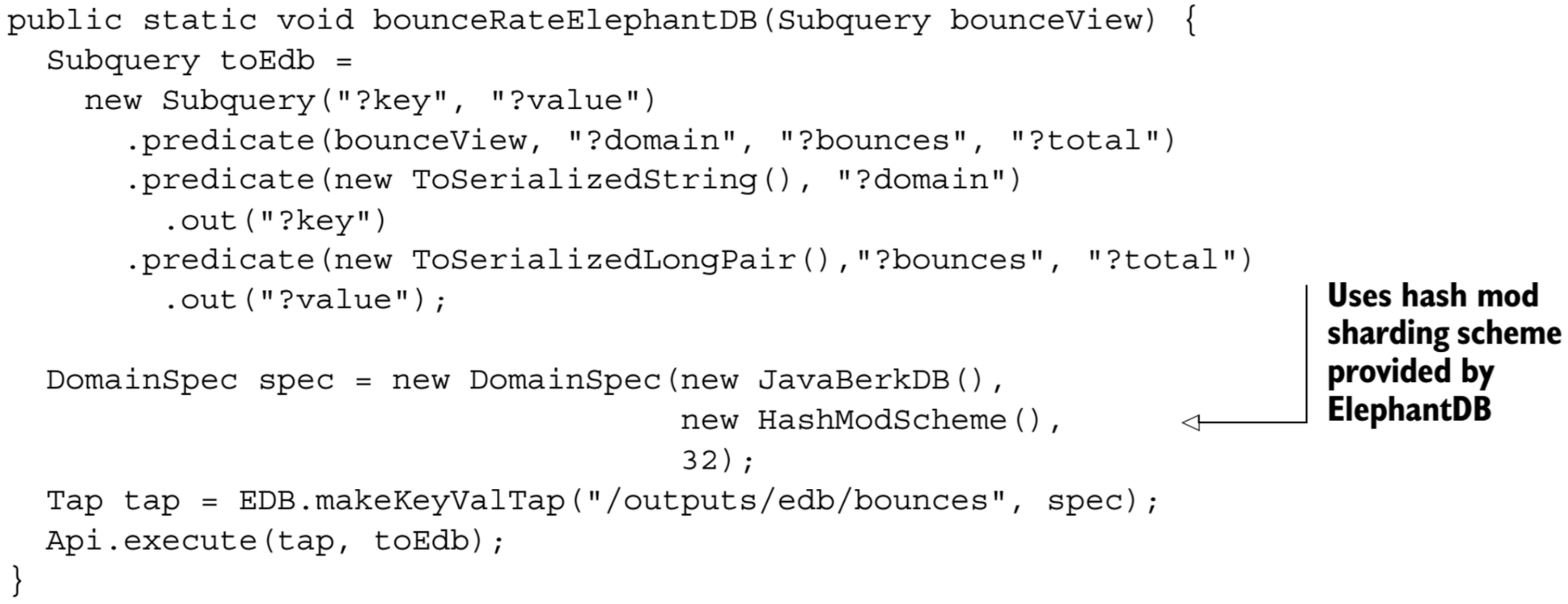
```
public static class ToSerializedString extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    String str = call.getArguments().getString(0);

    try {
      call.getOutputCollector().add(new Tuple(str.getBytes("UTF-8")));
    } catch(UnsupportedEncodingException e) {
      throw new RuntimeException(e);
    }
  }
}

public static class ToSerializedLongPair extends CascalogFunction {
  public void operate(FlowProcess process, FunctionCall call) {
    long l1 = call.getArguments().getLong(0);
    long l2 = call.getArguments().getLong(1);
    ByteBuffer buffer = ByteBuffer.allocate(16);
    buffer.putLong(l1);
    buffer.putLong(l2);
    call.getOutputCollector().add(new Tuple(buffer.array()));
  }
}
```

This serialization function is essentially identical to the one for the composite keys.

Allocates space for two long values

BUILDING THE SERVING LAYER FOR SUPERWEBANALYTICS.COM

- Bounce-rate analysis
  - ideal view is a key/value index



```
public static void bounceRateElephantDB(Subquery bounceView) {
  Subquery toEdb =
    new Subquery("?key", "?value")
      .predicate(bounceView, "?domain", "?bounces", "?total")
      .predicate(new ToSerializedString(), "?domain")
        .out("?key")
      .predicate(new ToSerializedLongPair(),"?bounces", "?total")
        .out("?value");

  DomainSpec spec = new DomainSpec(new JavaBerkDB(),
                                   new HashModScheme(),
                                   32);
  Tap tap = EDB.makeKeyValTap("/outputs/edb/bounces", spec);
  Api.execute(tap, toEdb);
}
```

Uses hash mod sharding scheme provided by ElephantDB