Data Virtualization Platform

© 2017 Alexandre Boudnik

Contents

[Preface 3](#_Toc471316349)

[Purpose 3](#_Toc471316350)

[Scope 3](#_Toc471316351)

[Architectural Goals and Constraints 3](#_Toc471316352)

[Assumptions & Speculations 4](#_Toc471316353)

[Components 5](#_Toc471316354)

[How it works 5](#_Toc471316355)

[Versioned WORM 5](#_Toc471316356)

[Partitions and Patchwork 6](#_Toc471316357)

[Data Blocks & Data Catalog 6](#_Toc471316358)

[Versioning 7](#_Toc471316359)

[As of Date Support 7](#_Toc471316360)

[Cache 7](#_Toc471316361)

[Retention & Eviction 7](#_Toc471316362)

[Data access 8](#_Toc471316363)

[Dependency Graph 8](#_Toc471316364)

[Functions 8](#_Toc471316365)

[Concurrency 8](#_Toc471316366)

[Invalidation & Lineage 8](#_Toc471316367)

[Scalability 8](#_Toc471316368)

[Security & Authorization 9](#_Toc471316369)

[Auditing 9](#_Toc471316370)

[Use cases 9](#_Toc471316371)

[OLTP 9](#_Toc471316372)

[OLAP / BI / Data Lake 9](#_Toc471316373)

[ETL/ELT 9](#_Toc471316374)

[DSS, What-if Analysis 9](#_Toc471316375)

# Preface

There are several problems that either missing in many COTS and FOSS platforms or requiring better support in virtually any data processing:

* Data state & persistency
* Versioning for data, data structures, code, and other metadata
* Uniform data access
* Multi-structured data
* Data lifecycle
* Granular data access rights & security
* ETL/ELT & Data Marts

The discussed architecture is capable to address many aspects of abovementioned problems.

# Purpose

This document provides a logical architecture for the Data Virtualization Platform (DVP), which is intended:

* To provide a conceptual design and to define interfaces to leverage different implementations
* To be used as a solution accelerator for wide range of applications including Data Warehouse, Data Lake, Event Processing, Fraud Detection, etc.

# Scope

The DVP is a platform responsible for the collection, storage, transforming data, and enabling data access for the purpose of Business Intelligence & Data Science, as well as for OLTP.

# Architectural Goals and Constraints

There are some key requirements and constraints that have significant impact on the architecture:

* The DVP will be based on a distributed architecture leveraging different deployment models for physical, virtualized, and cloud environment
* The DVP will need to support a robust and efficient message processing capability that is able to capture, transport and store a large volume of data from multiple physical data sources
* The DVP will need to ensure the protection of data from unauthorized access. All access must leverage approved identity access management requirements
* The DVP will need to provide security for the data at rest and data in motion, which means that DVP will ensure that Information Security policies are enforced within the DVP and in the internal and external communications
* The DVP will be highly dependent on network connectivity to the various data sources

# Assumptions & Speculations

Assume that there are two kinds of data:

* derived (or deterministic), which can be reproduced EXACTLY again and again, for example, results of pure function those doesn't charge the context
* primary (or nondeterministic), which are unique to our universe, and, in general, can't be reproduced, such as external data as of the moment in time, human input, and, in certain cases, randomly generated numbers

Using that discriminator, the only difference between these data is lifetime: primary data are persistent while derived data are ephemeral. Both primary and derived data are described and accompanied with versioned metadata, which are also immutable.

Therefore, all data are immutable and versioned, and thus, stateless. The stateless nature of data allows to significantly simplify application development by:

* eliminate any data concurrency issues
* leveraging functional programming paradigm
* increasing code re-use
* avoiding of complex ETLs
* on-demand extraction from multi-structured data

# Components



There many moving parts here, but main are:

* Persistent WORM (Write Once Read Many) memory, which contains:
  + versioned nondeterministic data
  + versioned metadata & versioned code
* Transient Cache, where deterministic data are stored
* Computational Engine, which gets data from WORM and cache, performs calculations, and puts results back in cache
* APIs for:
  + Metadata & Code access and management
  + Version management
  + Data Access for both primary and derived data (Function calls)

The outside world is represented by ESB, such as Apache Camel, that handles all inputs:

* External data sources
* Human input from any interactive UI client software

The ESB uses its own scheduler to call external sources or consumes data from streams.

# How it works

## Versioned WORM

WORM could be implemented as compound that may include the heterogeneous data sources:

* RDBMS like Oracle, Greenplum, or AWS RDS, accessible via jdbc
* NoSQL like Apache Cassandra or Apache HBase
* Storages for Ordered Sequences like Apache Kafka
* Local, shared, or cloud file systems with structured & plain files
* SDS (software-defined storage) like DDN’s WOS

WORM design resembles Query-Driven Warehouse design when:

* Unified access methods (wrappers/mediators) are built on top of heterogeneous data sources
* The metadata is used to translate the query into queries appropriate for individual data sources

To overcame high cost of queries in such design WORM should:

* Read less data by utilizing Partitions or Patchwork
* Send less data over network by leveraging Share Nothing, Collocation, and Local filtering

### Partitions and Patchwork

|  |  |  |
| --- | --- | --- |
| Related image |  |  |

Partitions in RDBMS have hierarchal organization: they are statically defined in DDL, and, to change the structure, they require data to be physically moved/copied.

Patchwork allows to query data by using arbitrary structure on top of squares (data blocks), dynamically built for that query without data movement: it is pure logical (speculative) concept.

### Data Blocks & Data Catalog

Data Block describes a quantum amount of data; it is a set of semantically similar objects, limited on some of their dimensions. It could be a file or directory on web, ftp, file system or HDFS. For example:

https://localhost/data/items.csv

ftp://localhost/data/orders/2016/

hdfs://pink-dumbo/data/

Or, it could be a valid jdbc URL with query containing pattern for prepared SELECT statement. For statement like:

select id,name,rate from salary where month='2016-12-01‘ and department=42

the encoded URL will look like:

jdbc:sqlserver://localhost:1433/AdventureWorks?select+id,name,rate+from+salary+where+month=$month+and+department=$department&month='2016-12-01'&department=42

Data Catalog is part of versioned metadata that organizes Data Blocks into a Patchwork – a functional equivalent to RDBMS catalog. Data Catalog contains Data Block URLs tagged with structured Key-Value tags where tag names divided into families:

* Data model attributes (optionally typed fields)
* Dimensions of that piece of data (limitations for column values, like ranges and lists)
* Technical attributes:
  + Signature (MD5 or SHA1) and check sum (CRC32)
  + Version, Producer job ID, Dates, etc
* Whatever user wants

This design provides flexible enough structure for easy querying.

### Versioning

There are many ways to represent versions in data.

One is more suitable for RDBMS where each record has two stamps: one is transaction ID (presume that sequence of transaction IDs is global and monotonically increases id values), which has created the record, and another is transaction ID, which deletes the record. There is no explicit UPDATEs: it represented by sequence of DELETE and INSERT.

Other is used in NoSQL where timestamps are in all records. There are no explicit DELETEs; they use special TOMBSTONE object as a value in deleted keys.

The Time Series is the particularly interesting design pattern. In addition to timestamp, dimension and value, we have to add one more temporal value – UPDATED.

## As of Date Support

To get business value of versioning, the session may contain session-wide variable asOfDate, which, if present, is used to control behavior of Data Access method to return data as it were on given date. Since metadata are also versioned, the metadata and code of corresponded version of are used.

## Cache

In opposite to WORM memory, the cache is transparent and transient by its nature. Cache is used to hold function results, which then used instead of actual calls. It is Key-Value store where:

* Key is a serialized function call (serialized method followed by the list of serialized actual parameters)
* Value is a serialized result either scalar or result set, followed by technical statistics for eviction

## Retention & Eviction

Cache Entry contains value, which accompanied with some statistics that helps to decide which entries should be evicted due lack of free space. Statistic may include:

* last time value was accessed
* access frequency
* dependency depth
* resources like CPU and IOs spent

When some value has sentenced for eviction, the whole dependency graph, which was based on it will be evicted as well. Obviously, the eviction caused by function call should only invalidate Data Blocks and Cache Entries outside of its Dependency Graph.

## Data access

All data access for both WORM and cache data are made through special getters, which stores the history of all accesses. The history of data access is used to build a complete dependency graph for the call.

## Dependency Graph

The history of data access could be compacted: a thousands of individual accesses to the sequential data elements could be replaced with a reference to Data Block. Dependency Graph is used to determine access rights for primary and derived data, as well for invalidation and lineage.

## Functions

During the function call it first looks in cache for the result; if found it updates the statistics and returns the value; if not, it calls the function, updates dependencies, stores the value in cache and returns result.

There are some aspects of Functions management:

* Lifecycle – becomes very simple since all pieces of code have been versioned
* Execution – proxies are used to ensure that only authorized and authenticated functions get through and the amount of traffic for a given function is limited or controlled
* Catalog – all function codes are stored along with reach metadata that ensures that developers of new solutions can find and re-use existing functions

## Concurrency

There are no such things as concurrent access for the primary and derived data, since all instances of Data Blocks and Cache Entries are versioned. However, Cache Entries destined for eviction may be used by active function calls. The simple reference count based garbage collection strategy will effectively finalize the eviction.

## Invalidation & Lineage

When new version of data or code have arrived, the function calls on them will be actually performed, building the entire dependency graph.

## Scalability

If distributed Key-Value or SDS are used for WORM implementation, just put more boxes there.

If In-Memory Grid is used for cache implementation, just put more boxes there.

If In-Memory Grid is used for calculation engine implementation, just put more boxes there.

## Security & Authorization

The system uses simplified Object Level Permissions access model. The Actors, Objects, and Methods make a dimensions of Access Matrix. In our case we don’t have destructive methods, which means that the only primary data may have explicit access limitation. All derived data have inherited (via dependency graph) limitations, which makes unauthorized access impossible. Some application may request to release their results to others: they have to call authorization service to explicitly remove the access limitations.

To protect data at rest and data in motion, 100% encryption could be turned on either for entire deployment or for certain types of objects.

## Auditing

Obviously the Audit Log is primary, nondeterministic data, which is stored in WORM. It is a historical perspective of all data accesses (actual or conceptual) and function calls. Since both access and invocation are made through getters and wrappers, it is easy to controls audit level.

Because of all abovementioned, the auditing requirements are trivially satisfied.

# Use cases

## OLTP

## OLAP / BI / Data Lake

## ETL/ELT

## DSS, What-if Analysis