LineageDB Architecture for real-time Big Data Analytics

A 10,000' tour of a Contemporary Data Analytics Platform

LineageDB's architecture background

Design and development of distributed systems (transactional + analytical)

- SOA + Minimizing Coupling
- Best-of-breed Polyglot topologies (all of the eggs are not in 1 basket)
- Multi-tenancy SaaS (internal and external facing)
- High-availability; Business Continuance; Disaster Recovery

Enterprise Integration

- Data Governance and Master Data Type Management (various ontologies & taxonomies)
- Integration of source types into Canonical Data Model
- Support of multiple business channel perspectives from the same book –of-record
- Command Query Responsibility Segregation (CQRS) vs. ETL
- Replication

Archives

- Immutable types (no destructive updates; no literal deletes)
- On-line analytics of 7+ years business events
- SaaS (versioning and purging)

Large Scale Analysis

- Data Marts
- Enterprise DW (internal facing & external facing)
- Rules-based engines; Machine Learning; Predictive Analytics
- Move computation to data

What's real-time? How big is big data?

- More marketing terms than valid propositions
  - Reminds me of MIPS
- Prefer 'Somewhere near immediately'
  - User experience/Service API defines time – the sooner the better
  - A consistent rate is best
- Your time is real-enough when SLAs are not troublesome or bothersome
- Given too little time, how much data is too much data?
  - Algorithm is strongly determined by the physical volume of data
  - Complex event processing (don't know when, or how much, data will arrive)
  - Aggregates over hierarchical time-series data sets
  - Time is money – what's the price tag of that data in that window of time?
- How fine are you slicing time?
  - How many need access to that data in that slice in time?
- How big is the data set you need to slice?
  - Divide and conquer, but only when you have to
  - Shards, partitions – don't roll your own
  - Distribute computation where the source data is physically located
  - Resource Oriented access patterns over complex ontologies and taxonomies

RDBMS – the great bits

3 cheers for SQL transaction systems

- **ACID**
- **Write-ahead transaction log = point-in-time recovery**
- **Simple CREATE DDL, simple INSERT, UPDATE, DELETE DML**
- **Memory management sub-system**
- **Query Optimizer (< 4 table joins)**
- **Security roles and rules enforced at column-row intersection**

One-size does not fit all

- **Select your data service(s) based on your business domain**
- **Select your DSL based on your computation**
- **Interactive & declarative DSL, maybe not SQL**
- **Time to move on beyond Immon and Kimball RDBMS-centric data world view – this ain't the '90s any more**

RDBMS is not all things to all computations

The down-side of RDBMS for analytics

- Relational Data models are not isomorphic with business model
  - Data model is too abstract (Normalized; Star; Snowflake; SQL anti-patterns)
  - Business users find it hard to work with models that are not identical, or very similar, in form to their business model
- Dimension and Fact tables are notoriously hard to maintain (type 3 + changes)
  - Type 3+ very rarely supported in EDW, DW or Data marts
- SQL Anti-Patterns abound in both transaction and analytic data models
  - Hierarchical schema, trees, time-series, aggregates, object model, etc.
  - If CREATE DML is valid then the relational server will create schema
  - If query DML is valid, the server will not optimize query but will run query
- B-tree does not scale linearly ($\log_2$ N at best)
- SQL is not best choice for analytic processing/algorithms
  - The simplest SQL queries are the best – they can be optimized
    - Anyone can learn to write SQL queries that only a SQL parser can understand

LineageDB is a Best-of-Breed Polyglot Enterprise DaaS

- CQRS
  - JSON (Avro; XML)
- ETLs
  - Software engineers, not ETL tool (Sqoop; Pig; MapReduce; Python)
  - DAG orchestration (Ozzie; Tez; Cascading)
- Key-value Data Storage Service
  - Cassandra (Riak)
  - alt. Columnar Service (HBase; Redshift)
- Index Service
  - ElasticSearch (Lucene)
- Graph service
  - Neo4j (Titan; GraphX)
- Messaging
  - Kafka (Camel; RabbitMQ) + ZooKeeper
- SQL Service
  - Hive (Qubole)
- In-memory Data Service
  - Storm (Spark Streaming) + ZooKeeper Spark (GridGain)
- Document Service
  - MongoDB (CouchBase)
- Distributed File Service
  - HDFS (S3)
- Business/Presentation Layer
  - Node.js + Angular.js + D3.js
  - Reporting tools
- RDBMS Service
  - Teradata; Oracle; MS SQL
- Resource Oriented Service
  - RESTful + cURL

The approach

Days can be spent discussing any one of the constituents of the LineageDB

The PaaS movement is changing how the data analyst and data scientist will interact with data services

So since we have an hour to cover a topic which is broad in scope, and one that many may be unfamiliar with, let's be practical

Limit coverage of LineageDB scope to CQRS through In-memory data services

- We won't be covering Document services through Resource-Oriented services.

Start with the data acquisition process that is common to all analytic architectures, but with a contemporary spin

Move onto subsequent constituents of the architecture, introducing technologies and tools along the way

This is not a Data Hub - architecture is not RDBMS centric

This is not a Data Lake - architecture is Data-as-a-Service

DDD + BDD + TDD

If you don't know where you're going any, road will take you there

An architecture that lacks a methodology is just a list of technologies.

Domain-Driven Design (DDD)

'Domain-Driven Design: Tackling Complexity in the Heart of Software,' Eric Evans

'Implementing Domain-Driven Design,' Vaughn Vernon

- Focus on the 'core' of the business domain
- Evolve models iteratively (see BDD + TDD)
- Master Data Types in core of domain
  - They can be found in un-structured as well as structured data
  - Their properties
  - Their relationships within the core domain
  - Their behavior within the core domain
- Governance of master data types
  - Data quality is a consensus settlement

Verify the story line - Behavior driven development (BDD)

User story represents aspect/feature of core domain

User story is clear and specific, and acceptance criteria is articulated

Validate the story line - Test driven development (TDD)

User story criteria tested pass/fail

Deliver user story every sprint – a DaaS is a software product, so build it as such

Git + Maven + Scala + Java (Python) + jUnit + ScalaTest + Cucumber + Jenkins + Chef (Puppet; Docker)

ETLs – look before you leap

The predominant technical obstacle to successfully delivering data analytic solutions are those silo'd data source systems.

- **ETLs are never as simple as they appear**
  - **Source systems are undocumented black boxes**
  - **Requires reverse engineering of business model state transitions**
  - **This is not a simple problem of moving data from point A to point B,**
    - **At point B, you are decoding logical units of work that ran at point A**
  - **Highly error prone due to lots of guesswork**
  - **Difficult to test (you will trip over every bug track in source data)**
    - **Higher bug maintenance = higher ETL price tag = more time waiting**
  - **Development is time consuming and resource intensive**
  - **Brittle ETL processes running in production**
    - **Changes in source application break extraction process**
  - **Very high latency**

CQRS – cure the illness, don't treat the symptom

If you have the application source code, then adopt Command-Query-Responsibility-Segregation (CQRS)

'Enterprise Integration Patterns,' Gregor Hohpe

Capture each create, update, and delete command where business model transitions state, the moment they happen

- **No more silos, no more black boxes, no more reverse engineering**
- **Eliminate guess-work -> gain high productivity, high quality, high durability**
- **Capture the commands – queries are not captured.**
  - **However, by capturing query command messages**
    - **Deploy SQL Injection Sentry**
    - **Construct Data Access Trails**

CQRS + MDT contracts

Master Data Type (MDT) Command Message Contracts

- A contract is a contract
- Very easy for business analyst and engineer to understand Message Contracts (JSON; Avro; XML)
  - Ubiquitous Language
- Easy to develop, unit test, complements continuously integrate, build,deploy
  - Append command message to existing class method
  - Configuration property files; Dependency Injection; Logging
- Contract definition casts source MDT definition into Enterprise Canonical Type
  - Conformance to Enterprise Canonical Model at the moment the source data is captured (staging goes bye-bye)
  - Business rules captured in Groovy, meta data & canonical model in JSON
- Much, much, simpler than transaction, or data, replication
  - No articles, publishers, subscribers, distributors, etc.

CQRS = Very low data latency

Data acquisition happens at the source business application, as it creates, updates and deletes MDT instances

- Source data can be un-structured as well as structured
- Pick a messaging service and a streaming framework
- Queries at the source system are out-side scope

1. **At the moment the application transitions states, Master Data Type (MDT) command message is written to Message Service (Kafka; Camel)**
   - Message service provides persistent storage of command messages
2. **Storm spout pulls MDT command messages off message queue**
3. **Storm bolt transforms MDT command message, and writes MDT instance into key-value store, one at a time.**
   - Transformation bolt is an in-memory staging space
   - Transformation bolt dynamically references code libraries (which contain references to the history of)
     - Business Rules (Groovy) (Transforms; Data cleaning;)
     - Metadata (JSON)
     - Canonical Model (JSON)
   - Enforce Canonical Model transformations & loads at single API (distribute transformers/loaders)
4. **Chain of storm bolt transforms master data type command message, and writes to index service (1 bolt per index), one at a time.**

Delta ETL = High latency

Delta ETLs are Logical Units of Work that are orchestrated (DAG)

1. **Delta Extract process (Bash; Perl; SQL; Sqoop; Pig; Python; Java; Flume; LogStash; Cascading) reads collection of recently created/changed MDT instances from data source**
   - **Enable change data capture mechanism within source data service, else be prepared to roll your own.**
   - **Wrap specifics of source system delta events and outputs events to Delta Transform API**
2. **Delta records are input to Transform process which handles each instance in the collection and fabricates a MDT Command Message, and writes command messages onto queue (Camel)**
   - **Individual, mini-batches, batches of command messages**
3. **Loader process (Storm spout) pulls master data type command messages from queue, transforms (Storm bolt) the messages and writes instance (in canonical form) into key-value data store**
   - **Transformation bolt is staging space**
   - **Bolt dynamically references code libraries**
     - **Business Rules (Groovy; Ruby) (Transformation; Data Cleaning; Data Quality)**
     - **Metadata (JSON)**
     - **Canonical Model (JSON)**
   - **Enforce Canonical Model transformations and loads at single API**
4. **Loader process transforms command message and loads into index service**

Baseline ETLs = days, if not weeks

Whether CQRS is used or Delta ETLs are used, you still need to take a baseline of the data source.

Get all records from all core domain tables in data source (orchestration)

Do not re-create source schema in a staging storage service

1. Extract process (SQL, Perl; Bash; Sqoop; Pig; Python; Java; Cascading) reads all MDT instances from data source, writes to disk storage (HDFS; S3) (temporary; move to archival storage)
2. Extracted records are input to Transform process which handles each instance in the collection and fabricates a MDT command message, and writes command messages onto queue (Camel)
   - Transformation process dynamically references code libraries
     - Business Rules (Groovy; Ruby) (Transformation; Data Cleaning; Data Quality)
     - Metadata (JSON)
     - Canonical Model (JSON)
3. Loader process (Storm spout) pulls MDT command messages from queue, (Storm bolt) transforms the messages and writes instance into key-value data store in canonical form.
   - Individual, mini-batches, batches of command messages
   - Transformation process dynamically references code libraries
     - Business Rules (Groovy; Ruby)
     - Metadata (JSON)
     - Canonical Model (JSON)
4. Loader process further transforms message and writes into index service

Reconcilers – done means done

Every data source and LineageDB go out of sync with one another at some time or another

- Source system or LineageDB goes off-line
- Network outage
- Natural disaster

The function of the reconcilers is to bring the contents of the LineageDB into conformance with the contents of the data source system.

Periodically, you need to take a random sample of master data type instances in the data source system and compare those records to what you have in the LineageDB.

- Alternative to a random sample: take a snapshot of a window in time

If something is amiss, then run the corresponding reconciler(s)

- Reconciler process dynamically references code libraries
  - Business Rules (Groovy; Ruby)(Transformers; Data Cleaning; Data Quality)
  - Metadata (JSON)
  - Canonical Model (JSON)

RDBMS data store for data analytics

The traditional approach to data analytics

- **Tightly coupled to expensive RDBMS clusters**
- **Limited to star and snow-flake schema**
  - **Notoriously difficult to maintain**
  - **Not easy for business users to consume**
- **Heavily dependent on brittle, expensive, ETLs**

RDBMS can be scaled vertically (at big price points), but eventually you run out of run-way 'cause a b-tree does not scale linearly.

The morphing of RDBMS services into MPP appliances have resulted in platforms that are not flexible enough to support rapidly changing data analytic needs.

- **Set-up for a given analytic process is time-consuming**

Key-value data storage service

Key-value data store services

- Run on commodity hardware
- Scale horizontally
- Have a simple, highly versatile, easy-to-consume data model
  - Write the data set into a form that mirrors the way you want to read it
- Data model scales linearly
- Open-source products + service providers

Alt. columnar service may be a better match to your core domain.

Platform-as-a-Service (PaaS) put clusters into hands of every business

From a data analytic perspective, the contents of the key-value store is the *book-of-record*.

LineageDB key-value data store

Preserves the 'lineage' of every MDT record instance.

- Each instance is versioned, and immutable
- Much easier to design code, test, deploy than type 3+ slowly changing data ETLs for Dimension and Fact tables

From the contents of the LineageDB, we can:

- Load/Re-Load data marts
- Reconstruct the version of the facts known to the business, at any particular point in time (in either its historically accurate form, or in a currently valid form)
- Manages and preserves histories of MDTs,
  - Ability to bind business rules, metadata and canonical model to MDT instances
- Evaluate the effectiveness of a machine learning or predictive analytic algorithm
- Operational Data Reporting (write the way you will read)
- Provision public facing DaaS, as well as private DaaS
- Because you have the history, you can visualize the evolution of business events as they occurred, over time, as well as how they are in the present moment
  - Node.js + Angular.js + D3.js

Schema-less Master Data Types

In that the physical and logical definitions of a master data type evolves over time, the schema-less aspect of key-value data services proves to be a great benefit to engineering business intelligence solutions.

The business to change the representation of a master data type at will, without needing to re-cast records that conform to a deprecated representation.

- **You can't get that capability from an RDBMS**

From this LineageDB we extract any atomic fact we need to support our data analysis process.

- **An atomic fact can be composite of primitive types**

From within Canonical Model you can project business channel views of MDT, as its form evolved over time.

Core benefits of LineageDB

By storing the MDT as a key-value (hybrid) data structure, we can easily derive whatever data representation is best suited for a given data analysis algorithm.

Extremely difficult to derive alternative types from dimension or fact tables

Analytic capabilities are no longer limited to dimension and fact data structures.

Collections of master data type records can become linked lists, arrays, map/hash table/dictionary, JSON document, AVRO files, CSV files, normalized or de-normalized relation, etc., whatever data structure is ideal for the analytic algorithm

Collections of master data type records can be stored as files in HDFS, columnar schema, property graphs, etc., whatever data storage is ideal for the analytic algorithm

Of course, the intent is to bulk load these target data structures from the LineageDB wherever possible.

- **Simplify disaster recovery and business continuance SLAs**

Additional benefits of LineageDB

We can extensively scale the key-value data store horizontally using commodity servers, SSDs, disks, as well as virtual machines hosted in-house or in the cloud.

- **Data partitioning and clustering handled by data service**

Implement as a cluster of peer-to-peer nodes = a highly available data warehouse with robust business continuance.

In that the individual records are *immutable*, the LineageDB may prove suitable to support relevant audit or compliance requirements.

Data from transaction system that is being deprecated can be loaded into the LineageDB so that the data the transaction system captured continues to be available to the business.

Data Analysts & Data Scientists crave indexes

At any point in time, we don't know what data might be needed by the business at the present moment or at some future point in time.

Data analysts and scientists don't know what particular content they might uncover, or need to uncover.

Finding the information the business needs to support its decisions is a huge challenge.

What they desperately need is indexes, lots of indexes.

Whenever using an RDBMS to manage large data sets, in order for the RDBMS cluster to be performant, you must keep your inventory of indexes to the smallest possible volume.

- **Otherwise, maintaining those indexes robs you of CPU cycles and memory needed to support data analysis computations.**
- **This resource constraint is why you have to adopt heap structures and partitions, and refrain from indexes, as the data volume grows into the 100s of TBs inside the relational data services.**

Enter the Index Service

In order to discover the content of the master data types, indexes of the master data types need to be created.

In order to search the data content from numerous view points, and to do so quickly, indexes are an absolute must-have.

To provide those indexes, a dedicated and optimized index service, such as ElasticSearch is needed.

- Indexes and types are defined via JSON
- REST API (best-suited to hierarchical schema) + Java API

The data analyst is able to use standard cURL to easily search, track and explore the data via REST.

Compared to using SQL, cURL is far easier to learn and master - and more powerful than SQL

- You can easily embed (JavaScript, or Groovy) scripts that process the data in relatively sophisticated ways within the context of the search.

Instances of index services, executing within a bounded application context or virtual machine, can be bulk-loaded with key-value pairs data (managed by the Index service cluster or by the LineageDB cluster)

Relationships & their properties

It comes as a big surprise to most people to learn that the term 'relational' in RDBMS has nothing to do with relationships between data type instances, nor with properties of those relationships.

- **Foreign key constraints were an after-thought to RDBMSs, and don't work unless you enable them**
- **They are turned off in the vast majority of transaction databases.**
- **An RDBMS does not fully support the ability to explicitly model a master data type and its relationships**
  - **To other type instances,**
  - **To itself, as well as**
  - **The properties that define those relationships (direction; effective date),**

There is a wealth of insight into the business that can be obtained by modeling relationships and their properties.

Graph Services

Businesses are beginning to recognize the graphs that are present in their core domains.

- **Organizational Units and the human resources that work/contract within, and across, those units (hyper-graph).**

A property graph is isomorphic with the business model

- **It contains the things the business has identified to be core to the business, and their names are ubiquitous to the business culture**
- **It contains the relationships the business has identified as existing between those things that interest them.**
- **This makes a property graph easy to understand and easy to use**

Graph this - Nodes and relationships

To easily analyze relationships you need a graph service, e.g., Neo4j, Titan, GraphX, etc.

- Roll your own graph algorithm

A node, a.k.a., vertex, is a named collection of key-value pairs

A relationship, a.k.a., edge, is a named collection of key-value pairs, along with a direction property and the identities of both the start and the end nodes.

Just like key-value data models, graph models are easy to understand.

Moreover, graph models are readily visualized

Index service + Graph service

The combination of LineageDB and Index service provides the means to define your graph nodes and relationships independently of the graph service used to analyze those nodes and relationships.

The Index service contains (JSON) types that represent node key-value pairs, as well as types that represent relationship key-value pairs + direction + start and end node ids

A Storm bolt keeps each index type instance in-sync with the contents of the LineageDB

Instances of graph services, executing within a bounded application context or virtual machine, can be bulk-loaded with node types and relationship types data (managed by the Index service cluster)

SQL Service != RDBMS

Various data services, running in the cloud and on premise, support some version of SQL. That does not mean, however, that they are RDBMS.

- Unlike an RDBMS, a SQL Service may or may not
  - Support ACID
  - Have a write-ahead transaction log (which allows you to recover at a point-in-time)
  - Use a b-tree data structure to manage its data
  - Have a query optimizer (SQL DML converted to mapreduce tasks)
  - Memory management sub-system (crash, ops ran out of memory)
  - Support mutable data sets
  - Manage data storage
  - ODBC/JDBC class library (reporting tool; multi-threaded consumers)

RDBMS in a LineageDB platform

- Write data sets for the way they will be read.
  - Avoid multi-table joins
- The storage capabilities of the RDBMS may or may not be used to store data for extended periods of time.
- The RDBMS storage capabilities, CPU, or memory are no longer used to support transforming, cleaning, or staging of data sets. These process now run outside of the expense RDBMS.

SQL Service

Usually just a SQL parser which translates query into MapReduce jobs

- Compatible with common reporting tools
- Variety of 'ANSI SQL' syntaxes available
- Parsers SQL, does not optimize SQL
- Cassandra Query Language (CQL) supports read DML over single table (column-family) (no joins) that is SQL-like

A Storm bolt transforms the command messages and uploads the message content into HDFS, into an S3 bucket, CSV file, etc., from where the SQL Service can read the data set.

Schema (metadata about) tables and views are defined

- Schema is optimized to support read requests originating from the analytic processes.

Instances of SQL services, executing within a bounded application context or virtual machine, can be bulk-loaded with data sourced from a column-family, HDFS, an S3 bucket, local file system, etc.

In-memory data service

Here's where the plot thickens – PaaS, like money, changes everything

Spark is used to support in-memory batch processing

- Can do transformations and window operations
  - MapReduce functions and related functions are provided
- Well suited to support Baseline & Delta ETLs, as well as loading Index, Graph, Data Mart, Dimensions & Facts, Aggregations

Storm, like Spark Streaming, is used to support in-memory processing of unbounded stream data, one tuple at a time.

- Computations and transformation via Bolts
- Well suited to support MDT command message Transforms, Data Cleaning, Data Quality , and loads into LineageDB and Index service.

Storm Trident is used to support in-memory micro-batch processing of streams.

- Filters, functions, aggregations, joins

Questions?

You can unbuckle your seat belt and walk around the cabin