LineageDB Architecture for real-time Big Data Analytics

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

LineageDB's architecture ancestry

- Design and development of distributed systems (transactional + analytical)
  - Maximizing Cohesion + Minimizing Coupling
  - Polyglot topologies (all of the eggs are not in 1 basket)
  - Multi-tenancy SaaS
- Data Governance and Master Data type Management
  - Integration of source types into canonical data model
  - Support of multiple business channel perspectives from the same book –of-record
- Audit Trails
  - Immutable types (no destructive updates; no literal deletes)
  - Data Access Patterns (security)
- Archives
  - On-line analytics of 7+ years business events
  - SaaS (versioning and purging)
- Large Scale Analysis
  - Enterprise DW (internal facing)
  - DW as Consumer Service (public facing)
  - Rules-based engines; Predictive Analytics; Machine Learning
- Command Query Responsibility Segregation (CQRS) vs. ETLs
  - Replication
- DDD + BDD + TDD

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 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 strongly determined by 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
  - Resource Oriented access patterns

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 based on your business domain
- Select your DSL based on your computation

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)
- SQL Anti-Patterns abound in both transaction and analytic data models
  - Hierarchical schema, 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
    - How long, complex and incomprehensible does a predicate expression have to become before you recognize that you using the wrong DSL

LineageDB is a Best-of-Breed Polyglot

- 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
- RDBMS Service
  - Teradata; Oracle; MS SQL
- Resource Oriented Service
  - RESTful

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 and tools

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

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

Domain Driven Design + BDD + TDD

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

- 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
- Verifying the story line - Behavior driven development
  - User story represents aspects/features of core domain
  - User story is clear and specific, and acceptance criteria is articulated
- Validating the story line - Test driven development
  - User story criteria tested pass/fail
- Deliver user story every sprint
  - 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**
  - **Highly error prone whenever you're on the outside looking in**
  - **Difficult to test (you will trip over every bug in source system)**
    - **Higher bug maintenance = higher ETL price tag**
  - **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 pattern

- **Capture create, update delete commands when/where business model state transitions**
  - **No more silos, no more black boxes, no more reverse engineering**
  - **We are not interested in the queries, just the commands**
- **Master data type Command Message contracts**
  - **Business analyst and dev/ops understand contracts (JSON; Avro; XML)**
  - **Easy to develop and test (append to existing methods)**
  - **Contract definition brings source master data type into canonical type**
  - **Much, much, simpler than transaction, or data, replication**
    - **No articles, publishers, subscribers, distributors, etc.,**

CQRS = Very low data latency

Happens as the source business application creates, updates and deletes master data type instances

- Source data can be un-structured as well as structured
- Queries at the source system are out-side scope

1. At the moment the application transitions states, Master data type command message is written to Message Service (Kafka; Camel)
2. Storm spout pulls master data type command messages off message queue
3. Storm bolt transforms master data type command message, and writes master data type instance into key-value store, one at a time.
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

1. Delta Extract process (Bash; Sqoop; Pig; Python; Java; Flume; LogStash; Cascading) reads collection of recently created/changed master data type 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 master data type 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
   - Transform & Load libraries to be used in CQRS as well as delta and baseline ETLs
   - Enforce canonical transforms 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.

1. **Extract process (Bash; Sqoop; Pig; Python; Java; Cascading) reads all master data type instances from data source**
2. **Extracted records are input to Transform process (Python; Java) which handles each instance in the collection and fabricates a master data type command message, and writes command messages onto queue (Camel)**
3. **Loader process (Storm spout) pulls master data type command messages from queue, (Storm bolt) transforms the messages and writes instance into key-value data store in canonical form.**
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)

Key-value data store for data analytics

The traditional approach to data analytics

- Tightly coupled to expensive relational data service 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.

In stark contrast, Key-value data store services

- Run on commodity hardware
- Scale horizontally
- Have a simple, highly versatile, easy-to-consume data model that scales linearly
- Open-source products + service providers

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

LineageDB key-value data store

Inside the LineageDB are stored every instance, of every version, of a master data type record captured from the source systems.

- Preserves the 'lineage' of every master data type record instance.
- Think in terms of type 3, 4, 5 etc., slowly changing data ETLs, but much easier to code, test, deploy

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

From the contents of the LineageDB, we can:

- Reconstruct the version of the facts known to the business, at any particular point in time
- Evaluate the effectiveness of a machine learning or predictive analytic algorithm
- Visualize the evolution of business events as they occurred over time, as well as 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 schema-less aspect of key-value stores enables 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 project business channel view of master data type, as its form evolved over time.

Core benefits of LineageDB

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

Unlike traditional relational data warehouses, analytic capabilities are no longer limited to dimension and fact tables.

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

Since it is implemented as a cluster of peer-to-peer nodes, we have 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.

Lastly, data from transaction systems that are being deprecated can be loaded into the LineageDB so that the data the transaction system captured continues to be available to the business.

A major challenge to the business is the tracking of any change to its underlying master data type records. The data analyst needs to see the information that was created as well as the changes it has undergone over time (within the source data systems).

The ability to readily observe such changes is critical to the successful development of machine learning algorithms - you have to be able to collect the evidence, recorded within the business data, to determine if the algorithm delivers the promised benefits.

Data Analysts & Data Scientists crave indexes

Data analysts and scientists care about the data itself, as well as what the data is telling them.

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.

The terrible secret about data analysis is that we 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 to every data analyst.

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 (Resource Oriented Service best-suited to hierarchical schema)

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 transforms the command messages related to node types and relationship types managed by the Index service, keeping 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
  - ODBC/JDBC class library (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 SQL Service may or may not be used to store data for extended periods of time.
- The SQL Service storage capabilities, CPU, or memory are no longer used to support transforming or staging of data sets. These process now run outside of the expense SQL Service.

SQL Service

Primarily SQL parser which translates query into MapReduce jobs

- Compatible with common reporting tools
- Variety of 'ANSI SQL' syntaxes available
- Parsers, does not optimize
- 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 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 loads into LineageDB and Index service

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 command message transforms , and writes 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