Catena project design specification

Version 0.1

Charles Duffy 08/09/2017

# Change log

|  |  |  |  |
| --- | --- | --- | --- |
| DATE | AUTHOR | VERSION | DESCRIPTION |
| 08/09/2017 | C.DUFFY | 0.1 | Initial version |
|  |  |  |  |

Table of Contents

[Overview 5](#_Toc493320069)

[Product Rationale 5](#_Toc493320070)

[Architecture 5](#_Toc493320071)

[Overview 5](#_Toc493320072)

[Parser 6](#_Toc493320073)

[Planner 6](#_Toc493320074)

[Executor 6](#_Toc493320075)

[Chaincode 6](#_Toc493320076)

[Client application 6](#_Toc493320077)

[Management 6](#_Toc493320078)

[Data structures 7](#_Toc493320079)

[Program data 7](#_Toc493320080)

[Chaincode invoke function argument structures 7](#_Toc493320081)

[Internal tuple representation 7](#_Toc493320082)

[Parse Tree 7](#_Toc493320083)

[Execution Plan 8](#_Toc493320084)

[System data layout 8](#_Toc493320085)

[Database system catalogues 8](#_Toc493320086)

[User data tables 8](#_Toc493320087)

[Statistics 8](#_Toc493320088)

[Permissions and Authentication 8](#_Toc493320089)

[Interfaces 8](#_Toc493320090)

[Chaincode invoke functions 8](#_Toc493320091)

[Parser interface 8](#_Toc493320092)

[Planner interface 8](#_Toc493320093)

[Executor interface 8](#_Toc493320094)

[Fabric client interface 8](#_Toc493320095)

[Detailed component design 9](#_Toc493320096)

[Executor 9](#_Toc493320097)

[Development plan 9](#_Toc493320098)

[Feature specification for prototype 9](#_Toc493320099)

[Data definition 9](#_Toc493320100)

[Data manipulation 9](#_Toc493320101)

[Permissions and security 9](#_Toc493320102)

[Front End Protocol 9](#_Toc493320103)

[Development plan 9](#_Toc493320104)

[Feature Matrix 9](#_Toc493320105)

Catena design specification

# Overview

*Catena* (development codename) is an SQL relational database utilising a Hyperledger Fabric[[1]](#endnote-1) permissioned blockchain as the storage layer. The entire database engine is implemented in Go as Fabric chaincode, meaning the engine executable image integrity is verified by the system and automatically distributed to all participants.

Catena is designed primarily as a distributed relational database server which provides all the additional advantages of blockchain technology to existing applications, with minimal porting effort. It should also provide an internal or embedded SQL access layer for other chaincode programs. Catena brings the ease of development and a standard, familiar environment to blockchain application developers.

# Product Rationale

**Better data access interface than currently available**

An SQL interface, even if just an embedded one, presents a more sophisticated interface for smart contract applications then the get-set style interface available in Hyperledger Fabric.

**Security**

Provides a cryptographically guaranteed log of all database operations. The whole code for the engine is verifiably replicated on the blockchain.

**Operational advantages**

Blockchain replication fabric makes replicated databases (relatively) easy; nodes can be swapped in and out with much less pain. Code upgrades are propagated via the blockchain itself. Parallelised operations can be achieved with multiple blockchain peers in ‘query-only’ mode reading from the blockchain. It is possible more advanced parallelised operations could be implemented using ledger-based messages to distribute plan fragments to processing peers and gather the results.

# Architecture

## Overview

Catena is implemented as a single chaincode executable. A nodejs Fabric client application provides a “listener service” for database applications connecting to Catena. The chaincode itself operates primarily via a single invoke function which accepts an SQL query string as an argument.

The database management system itself is structured fairly conventionally, following a parser-planner-executor arrangement. The executor component reads and writes data from the appropriate HF ledger.

WEB BROWSER

REST API SERVER (FABRIC CLIENT)

FABRIC CHAINCODE

BLOCKCHAIN

## Parser

The parser is takes a text string containing one or more SQL statements and produces a parse tree representing the query. The parse tree is a multiply linked-list of parse node structs. The parser is implemented using the Go ‘yacc’ utility and a hand-crafted lexical scanner.

## Planner

The planner takes a parse tree and emits an execution plan tree. The plan tree is a tree structure consisting of nodes and edges indicating tuple data flow from node to node. The leaf nodes are scan / write nodes which read and write data to and from the Fabric state data. Nodes in the plan tree may be scan nodes, filter nodes or aggregation nodes.

## Executor

The executor takes a plan tree, and executes it against the Fabric state. In the current design, a goroutine is spawned for each node in the plan tree. Goroutines send tuple data to each other via messages on channels. Goroutines corresponding to plan edge-nodes read data from the Fabric state, and transmit it to their plan parent nodes where the data is transformed and or joined, until the output tuple set is obtained. The output set is returned to the calling HF client application.

## Chaincode

The parser, planner and executor components are implemented as separate packages. The chaincode is a single Go executable containing all the components.

## Client application

A nodejs daemon implements a listening socket and handles incoming connections from database client applications. The nodejs daemon communicates with Catena chaincode via the nodejs Hyperledger client API.

## Management

# Data structures

## Program data

### Chaincode invoke function argument structures

### Internal tuple representation

A Tuple is an ordered list of data elements (a database row). Catena models Tuples internally as a wrapper struct containing Tuple metadata and an array of DataElement structs. A DataElement struct represents a single atom, ie, a single column value.

#### DataElement Struct

|  |  |  |
| --- | --- | --- |
| **Member** | **Type** | **Description** |
| ElemType | Enum | Enumerated type describing the datatype of this element. POC values limited to **integer** and **text** |
| ElemValue | Interface ElementValue | Interface containing a relevant Go value, which represents the appropriate database type |
| ColumnName | Text | Original column name from table scan, if relevant |
| ColumnAlias | Text | Alias of this element provided in query |
| ColumnOrder | Int | Order of this element from leftmost position (starting at zero) |

#### Tuple Struct

|  |  |  |
| --- | --- | --- |
| **Member** | **Type** | **Description** |
| TupleWidth | Int | Number of elements in tuple |
| TupleStructure | ElemType[] | Array of ElemType values in order of row structure |
| TupleElements | DataElement[] | Array of DataElement structus in tuple order |

### Parse Tree

The parse tree is a Go array of ParseNode objects. There is no encoding of the parse tree structure directly in the tree; the ParseNode objects are generic and can describe any node in the tree. Each contains an array of ParseNode objects to implement child-nodes.

### Execution Plan

The execution plan is implemented similarly to the parse tree, an array of ExecutionPlanNode objects.

ExecutionPlanNode objects are structured as follows:

|  |  |  |
| --- | --- | --- |
| **Member** | **Type** | **Description** |
| NodeID | Int |  |
| SourceList | [ Int ] | List of NodeIDs this node reads tuples from. Multiple in the case of join nodes |
| Target | Int | NodeID this node delivers tuples to. The special target of 0 means the output. |
| ActionType | Enum | The action that this node performs. May be **Aggregate**, **Function**, **Filter**, **Scan**, **Join**[[2]](#footnote-1), **Transform** |
| ProjectionList | [ TupleSpec ] | Specifies the tuple structure that this node emits |
| FilterSpec | ScalarExpr | Expression tree defining selection / filter predicate for this node. No filter if empty |
| ActionArgument | - | Action-dependent argument. |

## System data layout

### Database system catalogues

### User data tables

### Statistics

### Permissions and Authentication

# Interfaces

## Chaincode invoke functions

## Parser interface

## Planner interface

## Executor interface

## Fabric client interface

# Detailed component design

## Query Executor

### Program flow

### Functions

#### ReadTuple

#### SpawnExecutionTree

# Development plan

## Feature specification for prototype

Initial development targets a prototype which will support the following SQL RDMS features.

### Data definition

* Integer and Text datatypes only
* Table creation and deletion (CREATE TABLE, DROP TABLE). No constraints, domains, indexes or primary keys. No default values or sequences.
* System catalogues tracking relation name and attribute details.
* Single schema, “public”.

### Data manipulation

* SELECT statement support, limited to:
  + No subqueries (either correlated or not)
  + No joins
  + No common table expressions
  + Includes support for arbitrary expression list in WHERE clause
  + Includes support for ORDER BY clause
  + Includes support for SUM() and AVG() aggregates in select list
* INSERT statement support, limited to:
  + explicit VALUES clause only, no SELECT
* DELETE statement
  + Arbitrary value expression in predicate

### Permissions and security

* Single role, no permissions or authorisation

### Front End Protocol

* REST based query submission and simple HTML-formatted return values

## Development plan

## Feature Matrix

1. <https://www.hyperledger.org/projects/fabric> [↑](#endnote-ref-1)
2. Not supported in POC version [↑](#footnote-ref-1)