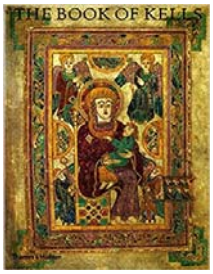
KEL

Peter M. Kogge



Background

- **KEL** = Knowledge Engineering Language
- Developed by David Bayliss, Lexis Nexis Risk Solutions Chief Data Scientist
- Target: ***Large scale graph analytics*** on ***large distributed systems***
- Outgrowth of prior ECL language
  - Enterprise Control Language
  - Built for Big Data processing (Hadoop on steroids)
- KEL Lite available for free download:
  https://hpccsystems.com/download/free-modules/kel-lite

Quote

- “KEL presumes that the user wants
- control over the logical data model, the analytic logic and the mathematics
- however, KEL also presumes that the user doesn't want to concern themselves with the details of
  - algorithm selection,
  - process construction,
  - the physical data model
  - or key building.”

KEL Properties

- Expresses graphs as tables
  - Vertices: Tables of **entities**
  - Edges: Tables of **associations**
- Both may have “Properties” – extra columns
- Declarative: specify what not how
- “Logic statements” specify properties of new subgraphs in Prolog-like combinations of old
- KEL tool chain creates parallel computation DAGs to pipeline data thru multiple steps

Sample Code: Jaccard

Define Graph

$$Persons := \mathbf{ENTITY}(...IntegerMyNeighbors, ...);$$
$$Addresses := \mathbf{ENTITY}(...);$$
$$ResidesAt := \mathbf{ASSOCIATION}(\mathbf{FLAT}, Persons, Addresses);$$
$$Gamma := \mathbf{ASSOCIATION}(\mathbf{FLAT}, Persons, Persons, Jaccard);$$
$$Persons := > MyNeighbors = COUNT(ResidesAt);$$

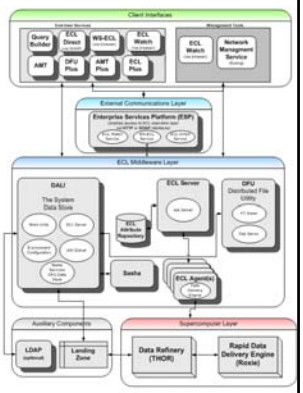
Define a new property

$$\mathbf{GLOBAL}: ResidesAt(\#1, \#3), ResidesAt(\#2, \#3)$$
$$=> Gamma(\#1, \#2, COUNT(\#3)$$
$$/(\#1.MyNeighbors + \#2.MyNeighbors - COUNT(\#3));$$

Assert what a new type of edge must look like

Lexis Nexis' Parallel Platforms

- Entity data kept in huge persistent tables
  - Often with **1,000s** of columns
  - Distributed over 100s of node
- Compilation process
  - KEL source transformed into ECL source
  - ECL transformed into C++ w' parallel libs
  - Parallel C++ runs in SPMD mode
- **THOR**: runs "offline" on 400+ node systems
  - Batch analytic processing over large data sets
  - Large distributed parallel file system
  - Leaves data sets for queries in indexed files
- **ROXIE**: runs "online" on smaller system
  - User queries using output files from THOR
  - Dynamically interrogate indexed files
  - Can perform localized computation



Software Architecture:
https://upload.wikimedia.org/wikipedia/commons/0/02/Fig4b_HPCC.jpg

Statement Types

- Entity Declarations: vertex classes
- Association Declarations: edge classes
- Property Definitions: define values to be associated with entities/associations
  - Extra "columns" in table representation
- Model Definitions: subsets of properties
- Input/Output of graphs
- Assertion Statements: define new subgraphs
- Query statements

Defining Graph Classes

Name of a vertex class

$$entity\_declaration \rightarrow \langle entity \rangle = \textbf{ENTITY}(filetype(prop\_list)\{,\textbf{MODEL}(prop\_list)\}?);$$

$$filetype \rightarrow \textbf{FLAT} \mid \textbf{XML} \mid \textbf{TEXT}$$

A list of properties and format for storage on external file system

Name of a vertex class

$$association\_declaration \rightarrow \langle association\_class \rangle = \textbf{ASSOCIATION}$$
$$(filetype\langle entity\_list \rangle \{prop\_list\}?));$$

$$entity \rightarrow entity\_classname$$

Typically two entity classes:
source vertex type
destination vertex type

$$property \rightarrow datatype?\ \{\textbf{UID} =\}?\ prop\_name\ \{= \textbf{NULL}(value?)\}?\ datafilefield?$$

$$prop\_name \rightarrow name \mid submodel$$

$$submodel \rightarrow name?\ \backslash\{prop\_list\backslash\}$$

Unique ID: property of a vertex class that uniquely identifies a vertex

Inputting Graphs

use_declaration → **USE** *file_name*(*filetype*,*entity_list*)

List of instance classes from which data in this file will be derived

Defines two classes of vertices and one class of edges

```
Actor := ENTITY( FLAT(UID=ActorID, Actor=ActorName) );
Movie := ENTITY( FLAT(UID=MovieName, Title=MovieName) );
Appearance := ASSOCIATION( FLAT(Actor Who, Movie What) );
USE IMDB.File_Actors(FLAT, Actor, Movie, Appearance);
```

Take the data from the IMBD database and extract from each row entries for Actor, entrices for Movie, and edges between them

Assertion Statements

assertion → (*scope*:)? *predicate_list*? => *production_list*

scope → **GLOBAL** | *entity*

production → *entity*(*argument_list*) **Asserts new vertex**

production → *propertyname* := *expression* **Asserts property must have some value**

- Predicate is a boolean expression over properties of vertices or edges
- Assertion defines logical relationships that a graph should have
  - If all predicates in predicate list are true,
  - then so must all productions
  - Even if that requires modifying graphs
- Predicates & productions use Pattern Variables as place-holders for values in actual entries

Example of Assertions And Pattern Variables

If someone has a Parent edge to someone else
Then there must also be an Ancestor edge

$$\mathbf{GLOBAL}: Parent(\#1, \#2) => Ancestor(\#1, \#2);$$

$$\mathbf{GLOBAL}: Ancestor(\#1, \#someone), Ancestor(\#someone, \#3) => Ancestor(\#1, \#3);$$

If someone is and ancestor of someone who is an ancestor of a 3rd person
Then the 1st person is an ancestor of the 3rd

Aggregates

- Using "$" as postfix on property in expressionsignals that some function to be applied to all matching entries
- E.g. $max, $min, $ave

Samples of Queries

$$query \rightarrow \textbf{QUERY} : queryname\{(argument\_list)\}? \ <= \ output\_expression\_list;$$
$$output\_expression \rightarrow \ entity \ \{(filter\_list)\}? \ \{\backslash\{projection\_list\backslash\}\}?$$

$$\textbf{QUERY} : \ AboveAveIncome \ <= \ Person(income \ >= \ income\$Ave)$$
$$\textbf{QUERY} : HighestIncome \ <= \ Person(income = income\$\hat{}Ave)$$

Sample Code: Jaccard

Properties

$$Persons := \textbf{ENTITY}(...Integer MyNeighbors, ...);$$
$$Addresses := \textbf{ENTITY}(...);$$
$$ResidesAt := \textbf{ASSOCIATION}(\textbf{FLAT}, Persons, Addresses);$$
$$Gamma := \textbf{ASSOCIATION}(\textbf{FLAT}, Persons, Persons, Jaccard);$$
$$Persons := > MyNeighbors = COUNT(ResidesAt);$$
$$\textbf{GLOBAL} : ResidesAt(\#1, \#3), ResidesAt(\#2, \#3)$$
$$=> Gamma(\#1, \#2, COUNT(\#3)$$
$$/(\#1.MyNeighbors + \#2.MyNeighbors - COUNT(\#3));$$

MyNeighbors property is out-degree of edge type ResidesAt

Gamma edge is defined to be between two Persons, with property =Jaccard coefficient

Execution Model

- Create DAG of Program – Bottom Up
- Start with entities needed by Query
- Id all statements that define instances of those entities
- Repeat up until the top
- ID USE statements that provide the data
- Create program that
  - Spreads USE data across all nodes
  - Streams that data thru DAG in parallel & pipelined
  - Save new graphs needed by QUERYs