Using Chapel for Natural Language Processing And Interaction

Brian Guarraci
CTO @ Cricket Health

Motivation

- Augment chat bot Human-created rulesets with data
    - ChatScript provides a powerful rule engine, but making Human-created rules is unscalable and limited
    - Use Chapel as a power-tool to create datasets which can be plugged into ChatScript engine
- Focus on two main types of custom datasets
    - Chord: Use word2vec for language support
    - Chriple: Use RDF triple stores for knowledge

Chord: Chapel + Word2Vec

- Word embeddings are vectors computed with a Neural Network Language Model (NNLM)
- Each word vector characterizes the associated word in relation to training data and other words in the vocabulary
- Vectors have interesting and useful NLP features
  - King - Man + Woman = Queen
  - Tokyo - Japan + France = Paris
- Replace Human-derived rules for certain NLP tasks

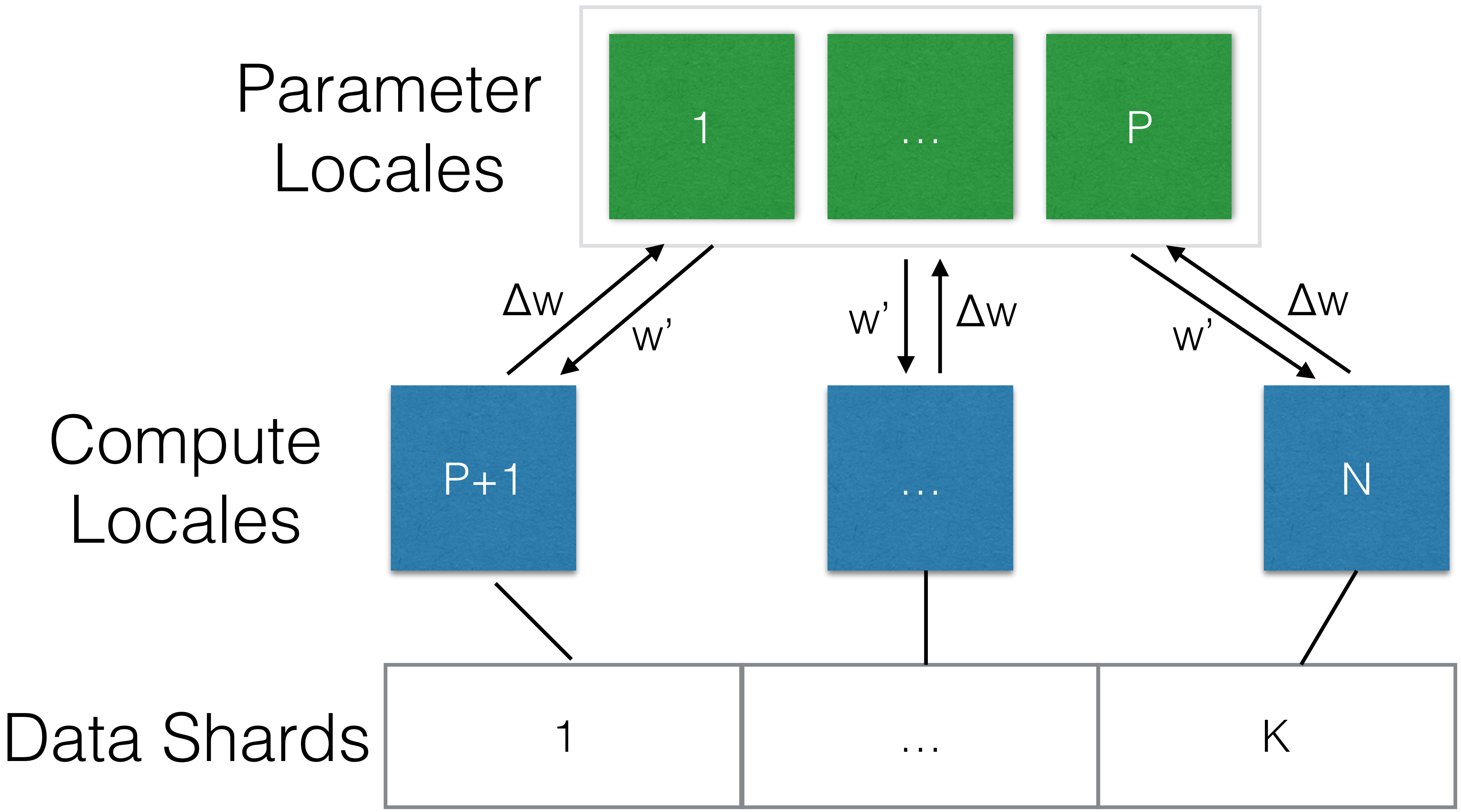
Chord: Path to Distributed

- First: Port Google's single-locale classic word2vec and validate
- Second: Port classic model to a multi-locale model
    - Maintain single-locale performance in multi-locale version
    - Preserve Asynchronous SGD (race conditions by design)
    - Encapsulate globals to ensure locale-local only access
    - Experiment with dmapped and other distributed memory strategies to find a fast method for cross machine data sharing

Chord: Path to Distributed

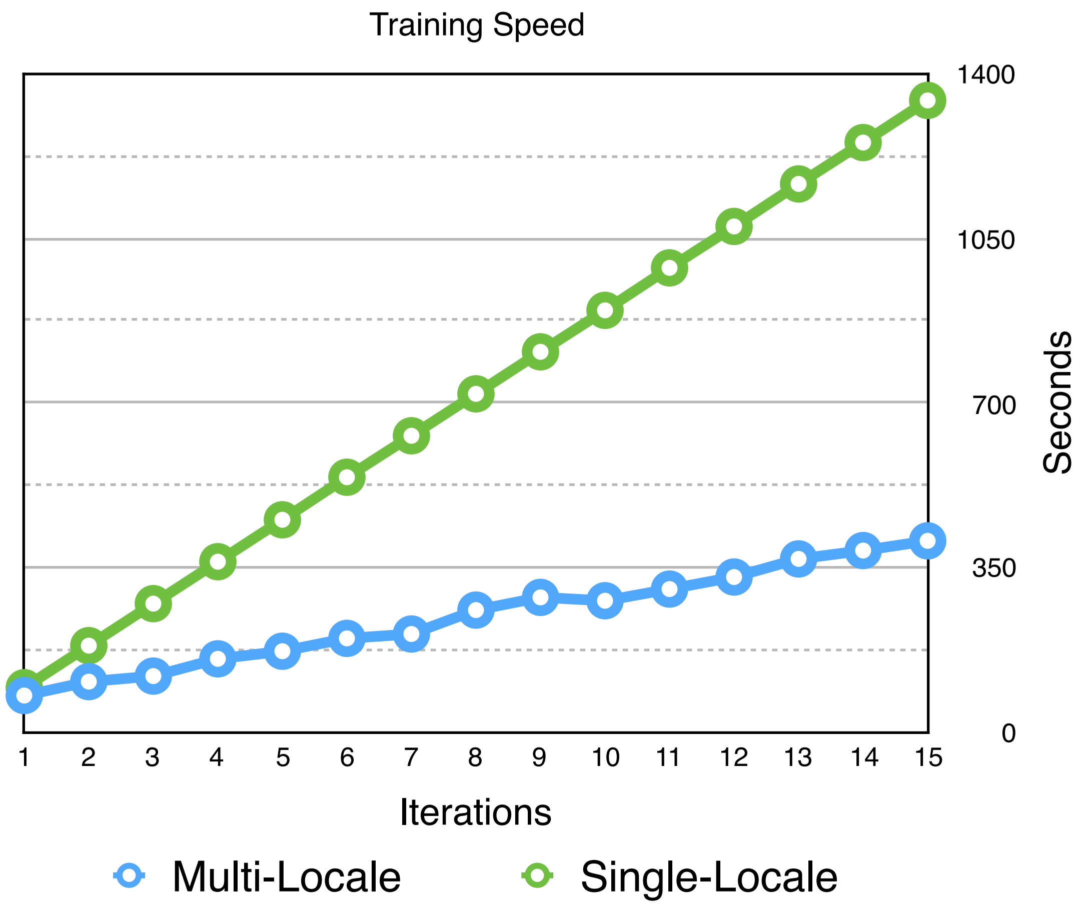
- Distributed models require periodic model sharing across locales
  - Naïve dmapped approach is very slow due to model specific behavior yielding excessive cross-machine data transfers
  - Use a variant of Google’s Downpour SGD
    - Reserve some locales as “parameter locales” and others as compute locales which train on data shards
    - Each compute locale diverges with it’s training data and updates the parameter locales after each training iteration
    - Use AdaGrad to perform model updates on param locales

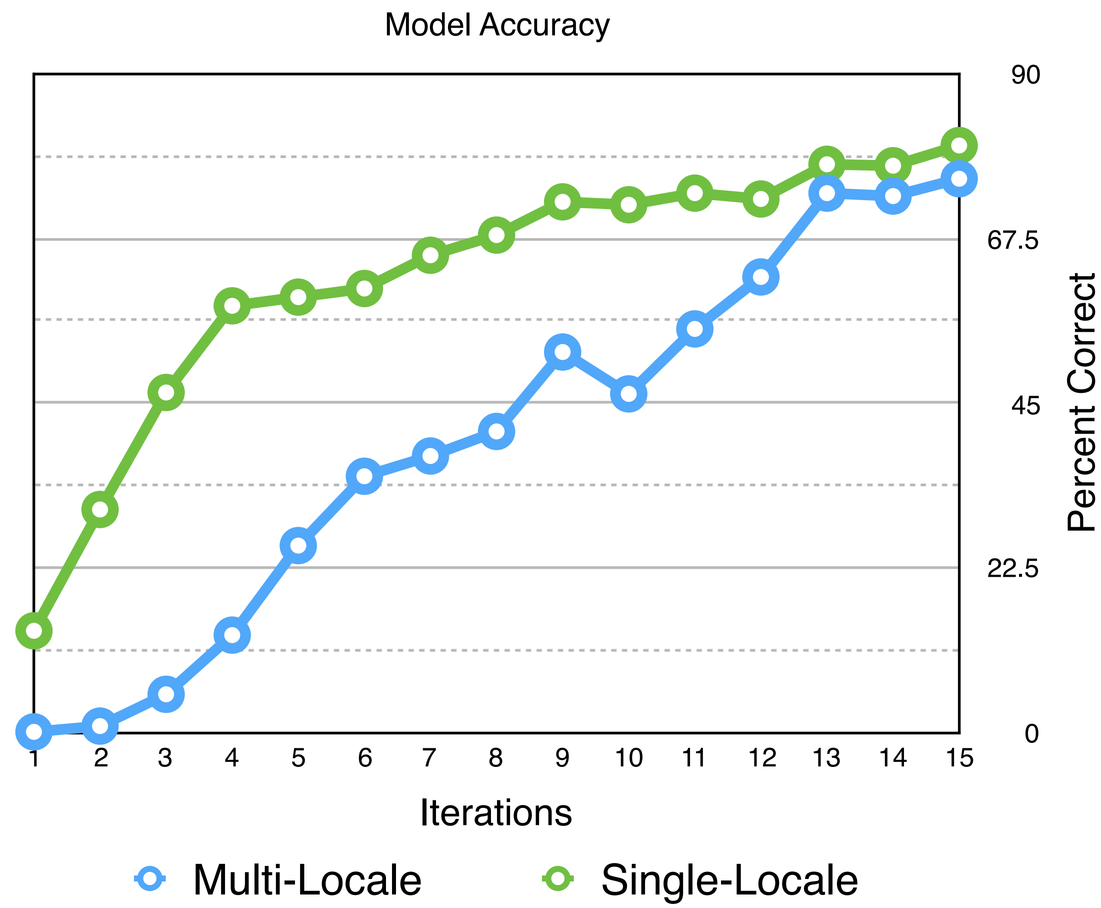
Chord: Architecture



Locales are partitioned into param and compute roles

Chord: Single vs Multi-Locale





Multi-Locale version > 3x faster with similar accuracy (eventually).

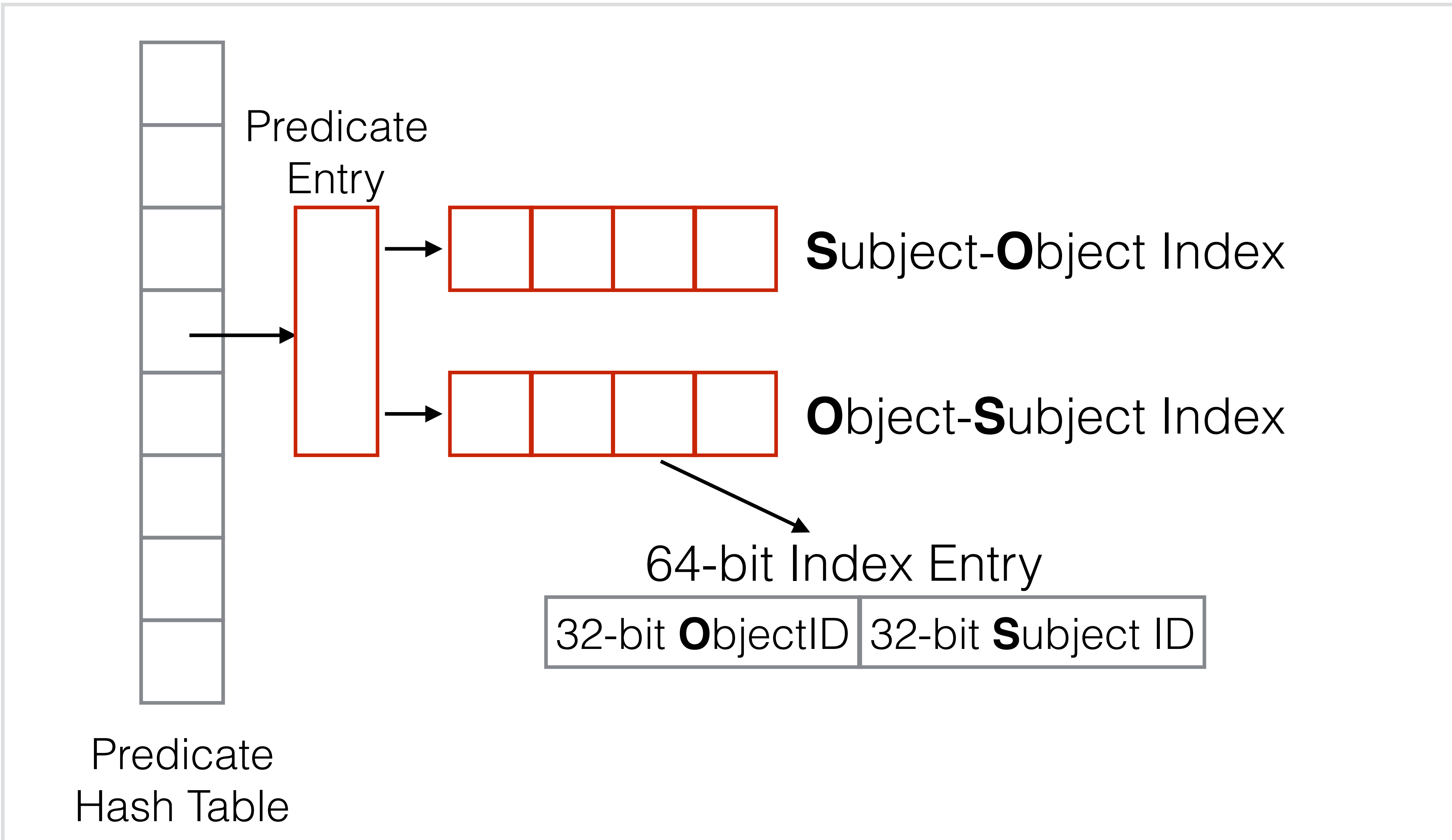
Multi-locale configuration:

- 8 locales: single parameter locale with seven compute locales
- Machine type: EC2 m4.2xlarge (8 vCPU 16GB RAM)

Chriple: Chapel + Triple Store

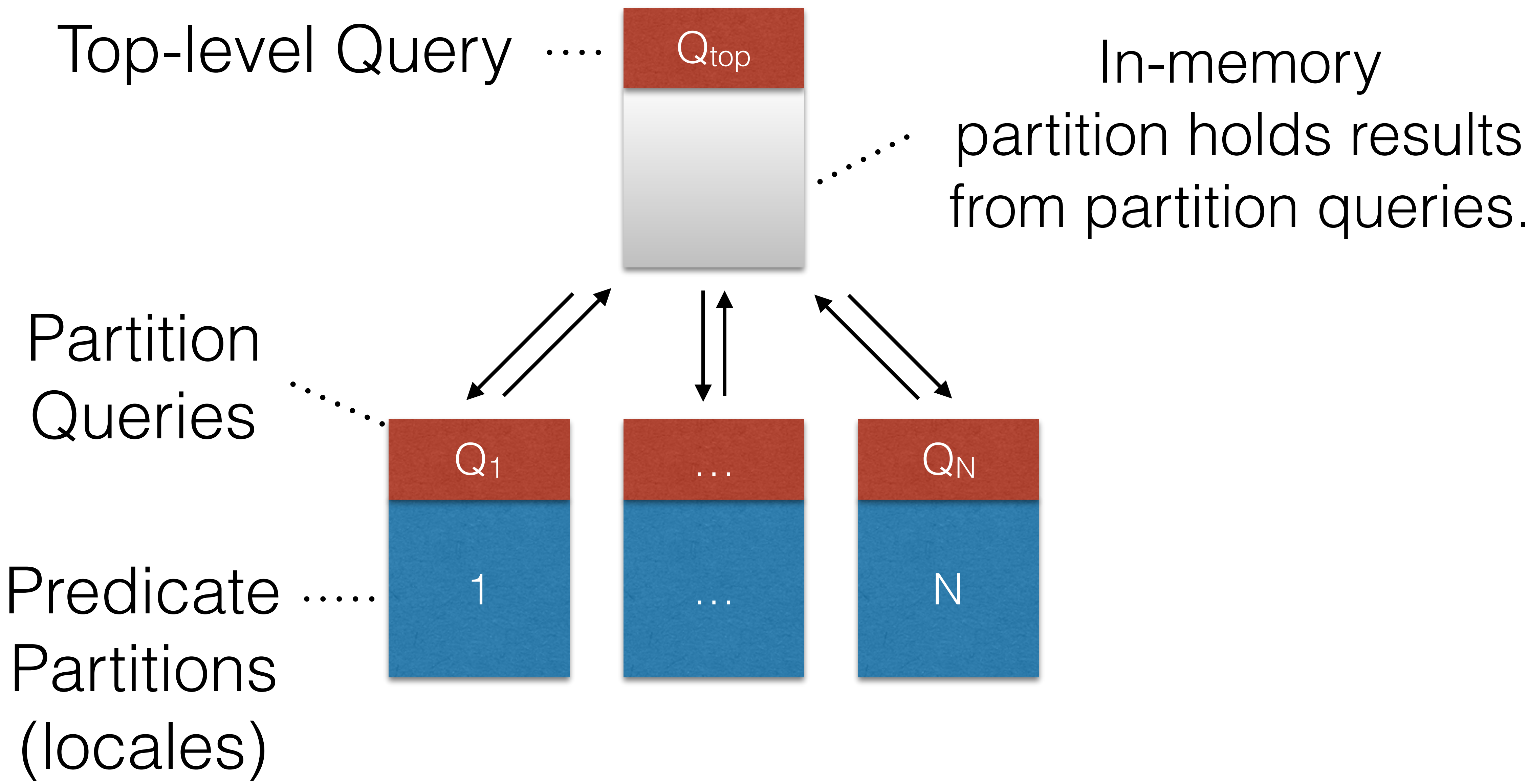
- Keep it simple to learn what's useful
- Naïve implementation inspired by TripleBit
    - Reasonably memory efficient
    - Predicate-based hash partitions on locales
- CHASM (from Chearch) stack-based integer query language
    - Supports essential distributed query primitives (AND/OR)
    - Supports sub-graph extraction

Chriple: Architecture



Locale Predicate Hash Partition

Chriple: Distributed Queries



Chriple: Current Results

- Memory requirements
  - ~16 bytes per triple
  - 2B triples require ~64GB RAM across cluster
- Performance (8 x EC2 m4.2xlarge [8 vCPU 32GB RAM])
  - 1.1M inserts / s (~137K / locale)
  - 40K reads / s [via parallel iterator] (~5K / locale)

AllegroGraph Benchmark

Load Test	# Triples	Time	Load Rate (T/Sec)
LUBM(8000)*	1.106 Billion	36min, 49 sec	500,679
LUBM(8000)****	1.106 Billion	48min, 30 sec	379,947
LUBM(160,000)*	22.12 Billion	12 hrs, 18m, 16s	499,188
AllegroGraph Pre-release**	310.269 Billion	78 hrs, 9m, 23s	1,102,737
AllegroGraph Pre-release***	**1.009 Trillion**	338 hrs, 5m	829,556

http://franz.com/agraph/allegrograph/agraph_benchmarks.lhtml

Conclusion

- Work in progress
  - Many opportunities for optimization
  - Useful for generating data and experimentation
- Code is available on Github
  - https://github.com/briangu/chord
  - https://github.com/briangu/chriple