Cassandra Explained

Disruptive Code
September 22, 2010

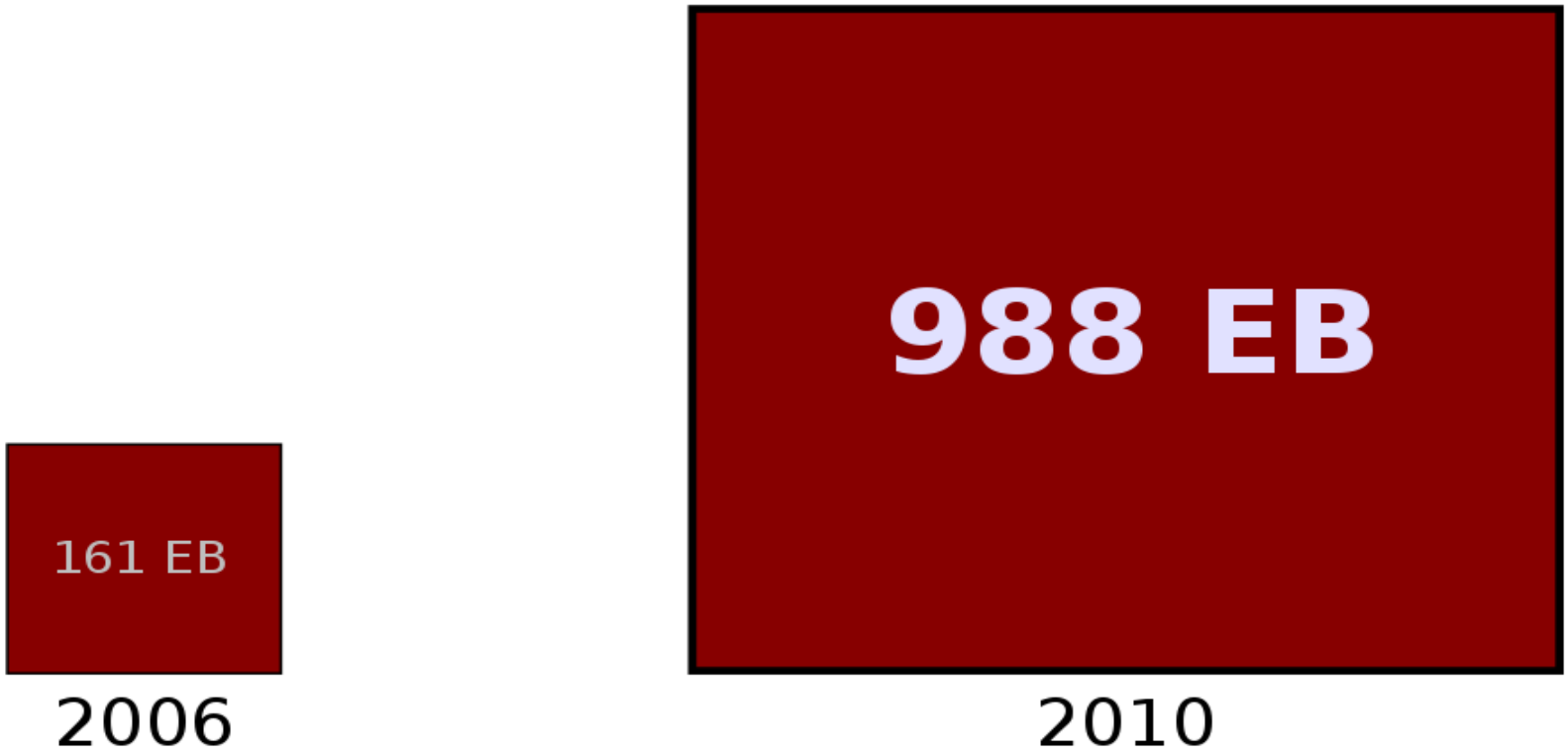
Eric Evans
eevans@rackspace.com
@jericevans
http://blog.sym-link.com

Outline

- Background
- Description
- API(s)
- Code Samples

Background

The Digital Universe

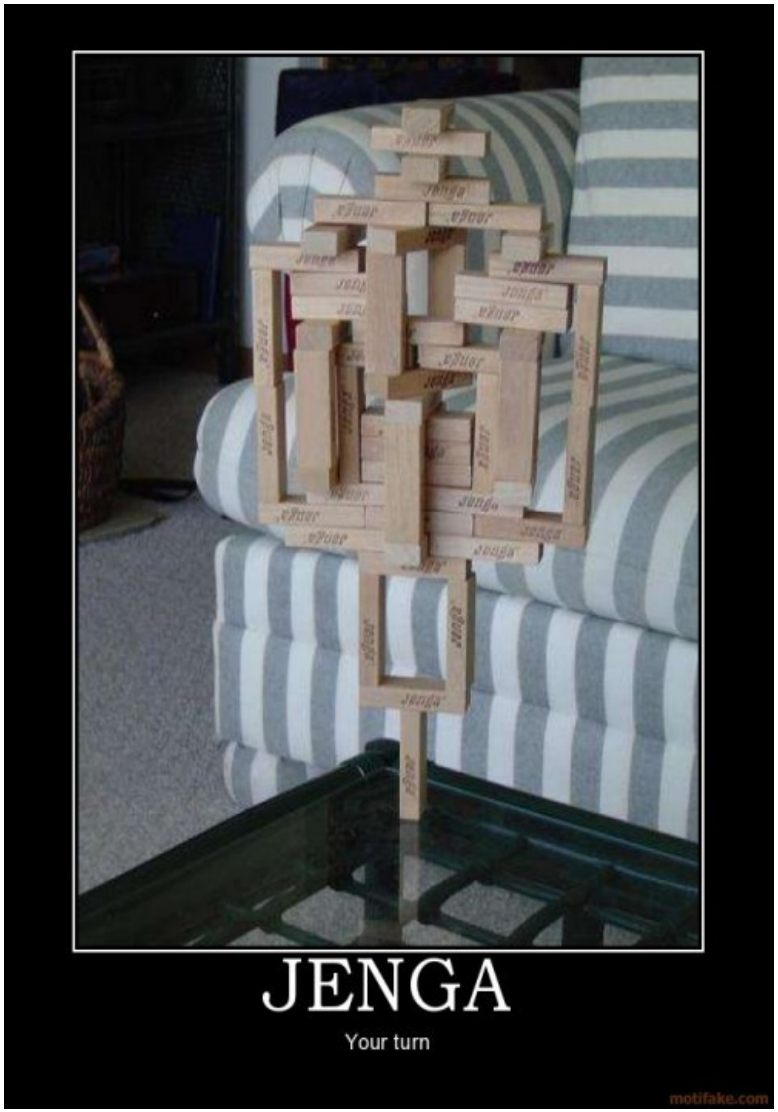


Source: IDC 2007

Consolidation

Old Guard

Vertical Scaling Sucks



Influential Papers

- BigTable
  - Strong consistency
  - Sparse map data model
  - GFS, Chubby, et al
- Dynamo
  - O(1) distributed hash table (DHT)
  - BASE (aka eventual consistency)
  - Client tunable consistency/availability

NoSQL

- HBase
- MongoDB
- Riak
- Voldemort
- Neo4J
- Cassandra
- Hypertable
- HyperGraphDB
- Memcached
- Tokyo Cabinet
- Redis
- CouchDB

~~NoSQL~~ Big data

- HBase
- ~~MongoDB~~
- Riak
- Voldemort
- ~~Neo4J~~
- Cassandra
- Hypertable
- ~~HyperGraphDB~~
- ~~Memcached~~
- ~~Tokyo Cabinet~~
- ~~Redis~~
- ~~CouchDB~~

Bigtable / Dynamo

Bigtable

- HBase
- Hypertable

Dynamo

- Riak
- Voldemort

Cassandra ??

Dynamo-Bigtable Lovechild

CAP Theorem “Pick Two”

- **CP**
  - Bigtable
  - Hypertable
  - HBase
- **AP**
  - Dynamo
  - Voldemort
  - Cassandra

CAP Theorem ~~“Pick Two”~~

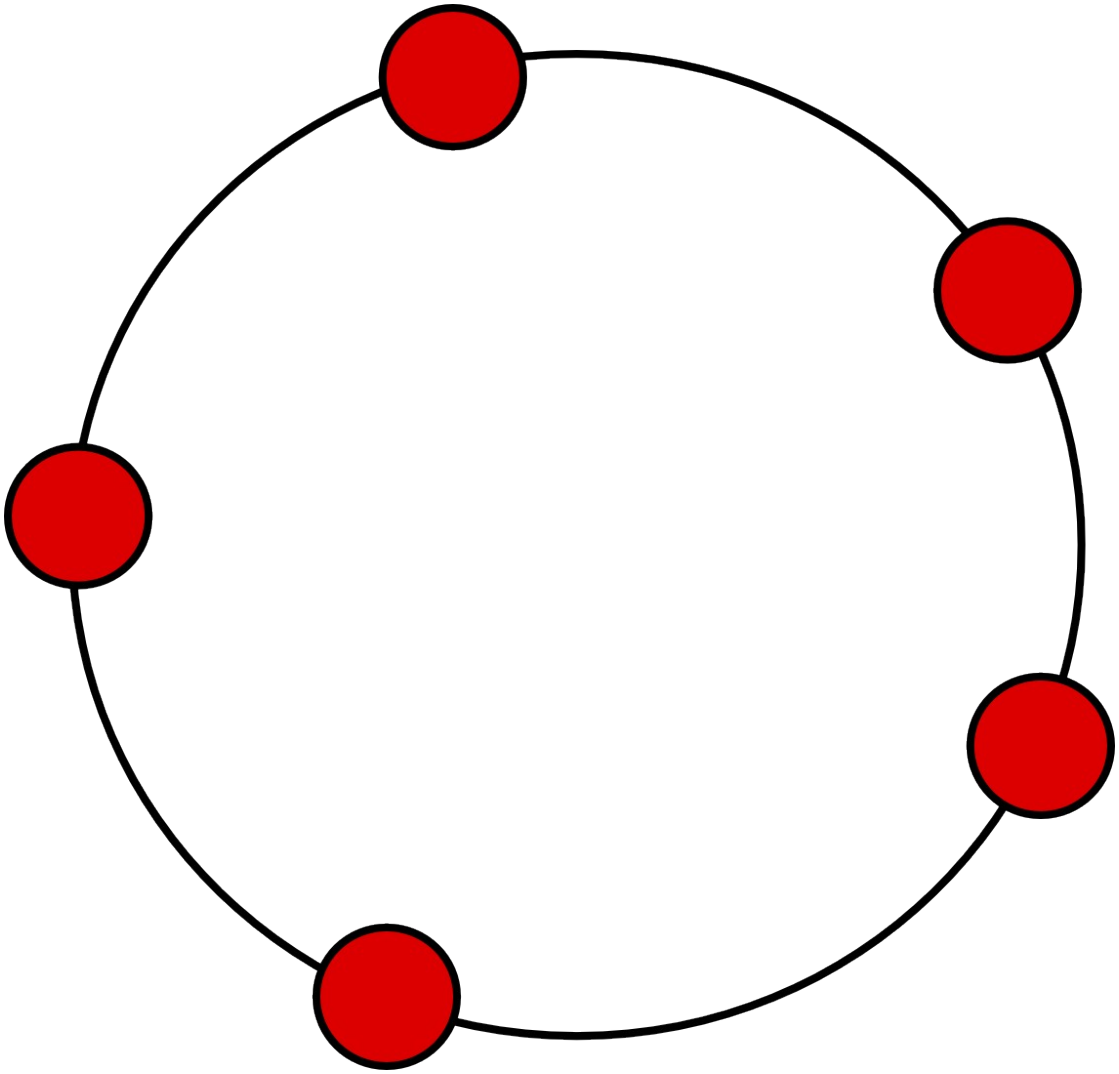
- Consistency
- Availability
- Partition Tolerance

Description

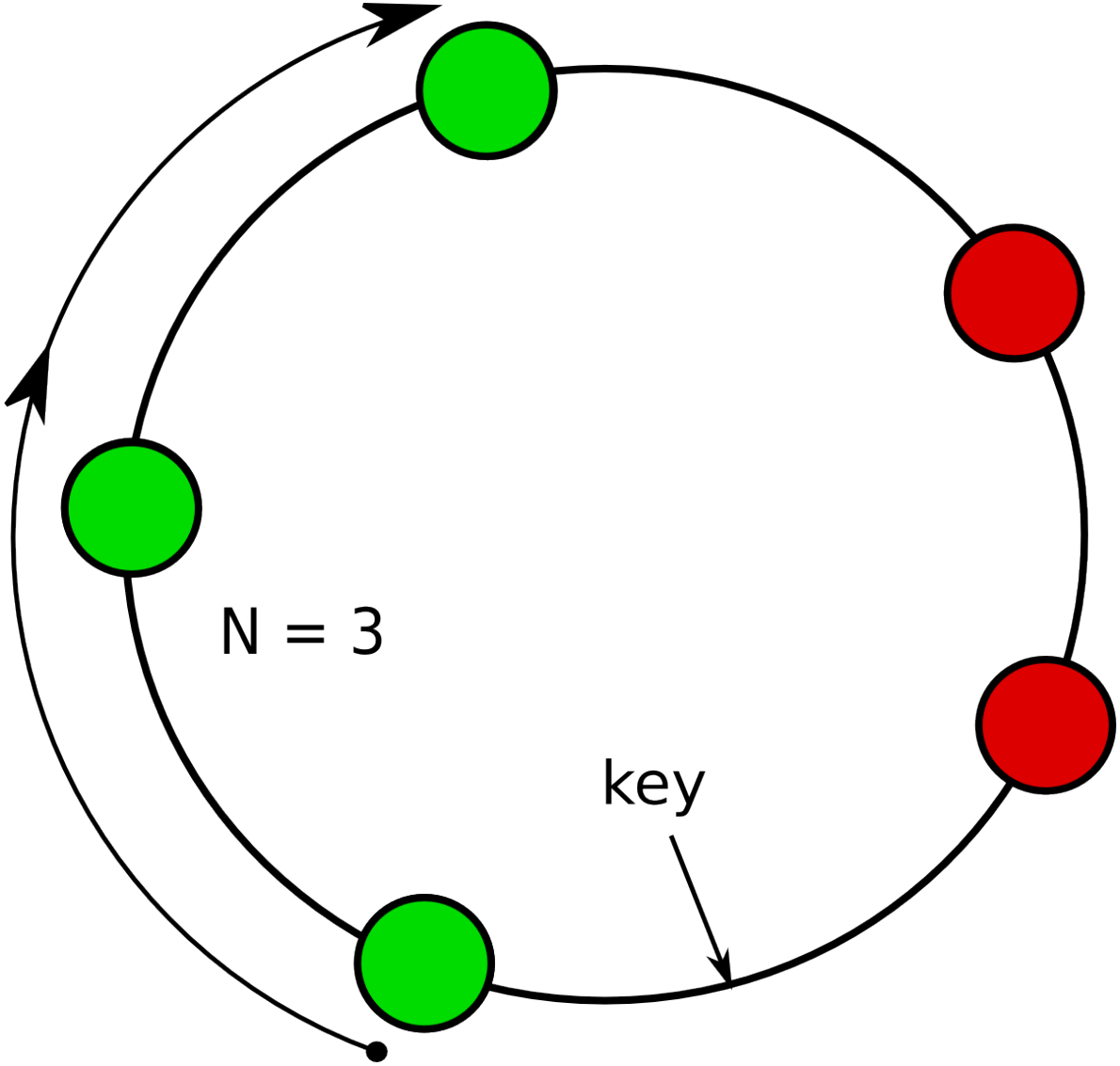
Properties

- Symmetric
  - No single point of failure
  - Linearly scalable
  - Ease of administration
- Flexible partitioning, replica placement
- Automated provisioning
- High availability (eventual consistency)

P2P Routing

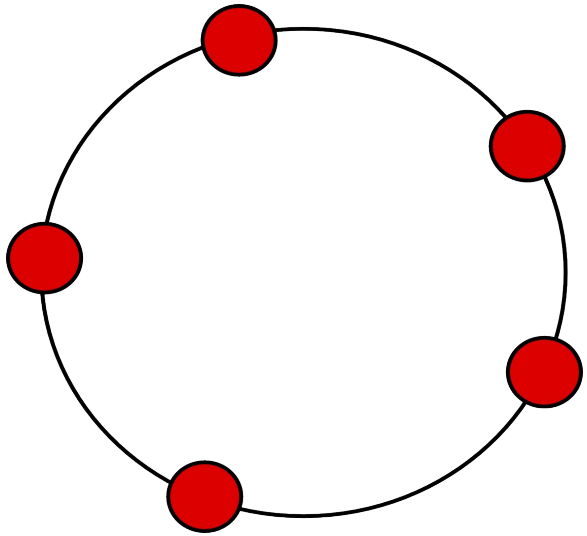


P2P Routing



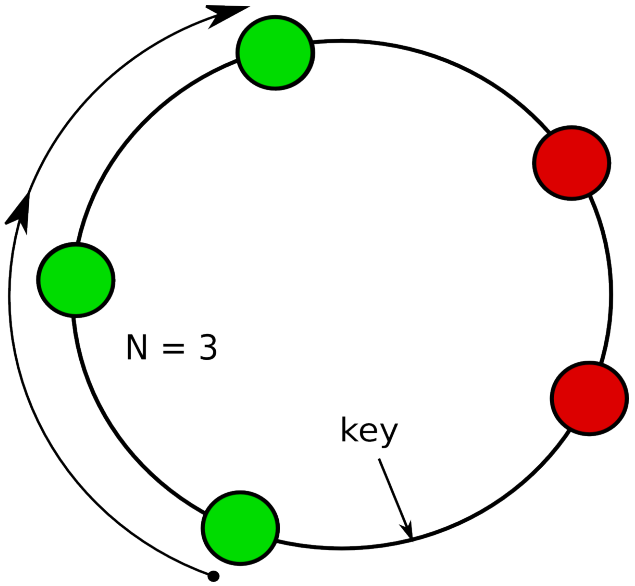
Partitioning

- Random
  - 128bit namespace, (MD5)
  - Good distribution
- Order Preserving
  - Tokens determine namespace
  - Natural order (lexicographical)
  - Range / cover queries
- Yours ??

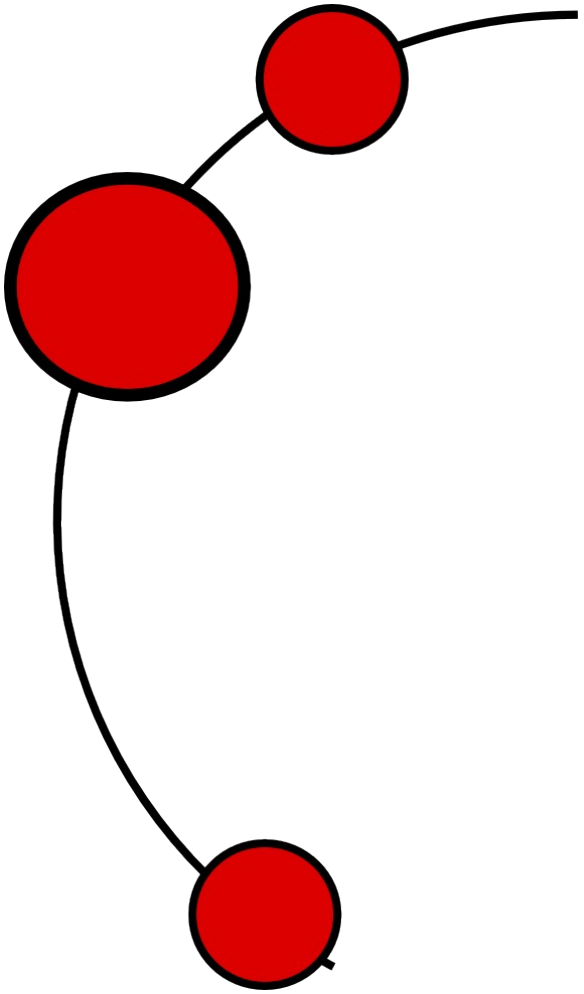


Replica Placement

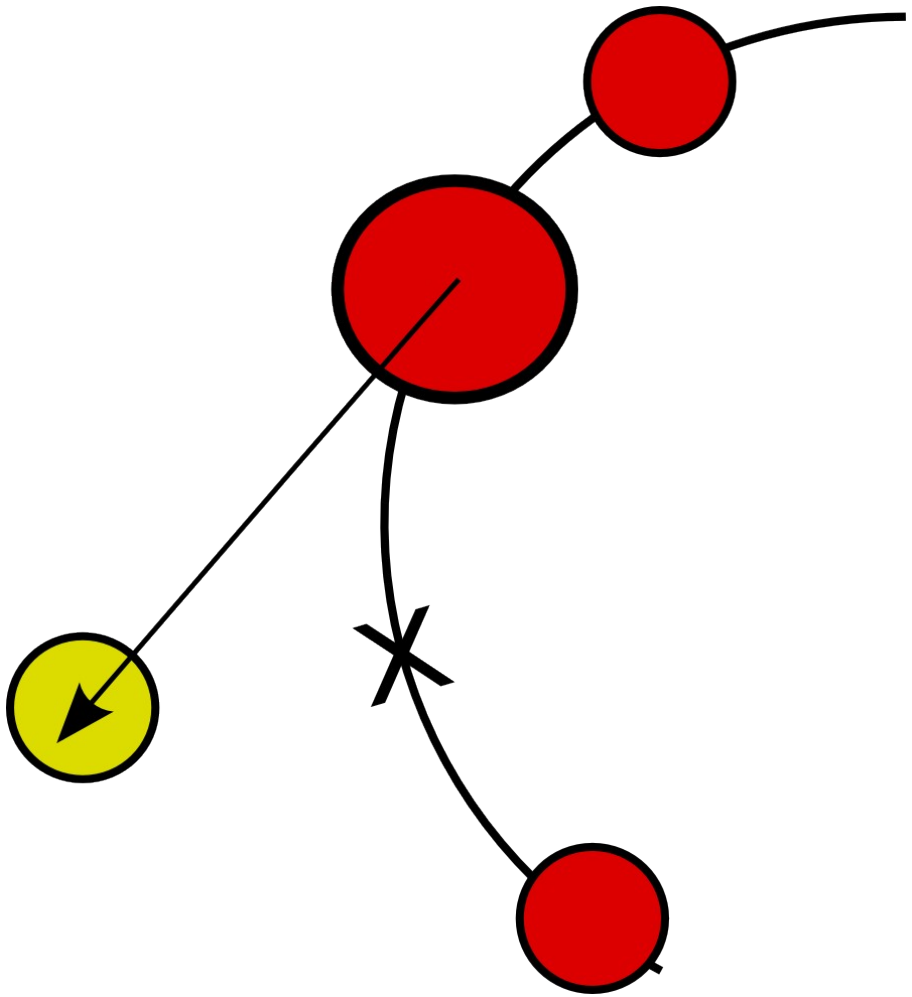
- SimpleSnitch
  - Default
  - N-1 successive nodes
- RackInferringSnitch
  - Infers DC/rack from IP
- PropertyFileSnitch
  - Configured w/ a properties file



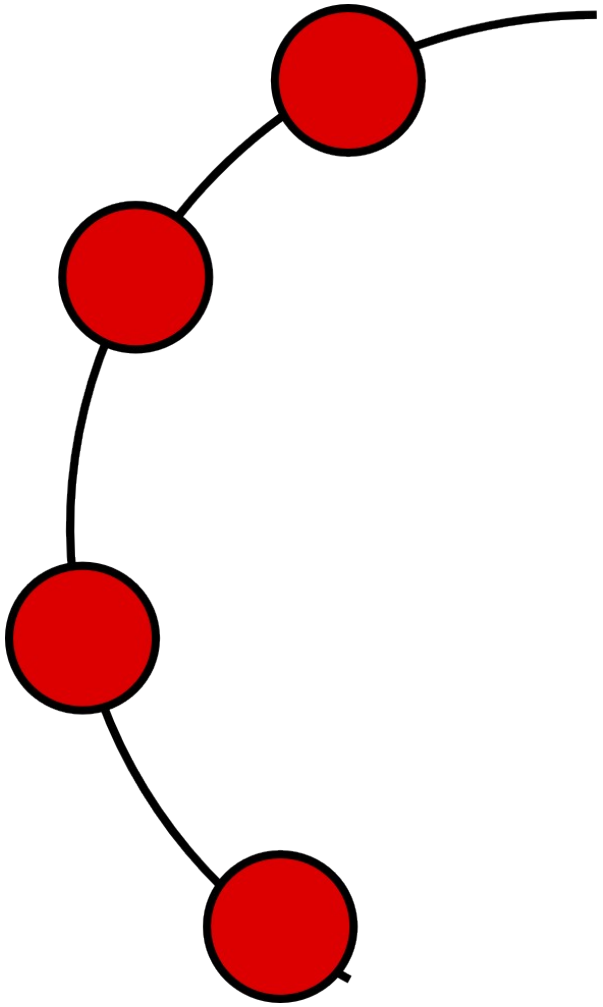
Bootstrap



Bootstrap



Bootstrap



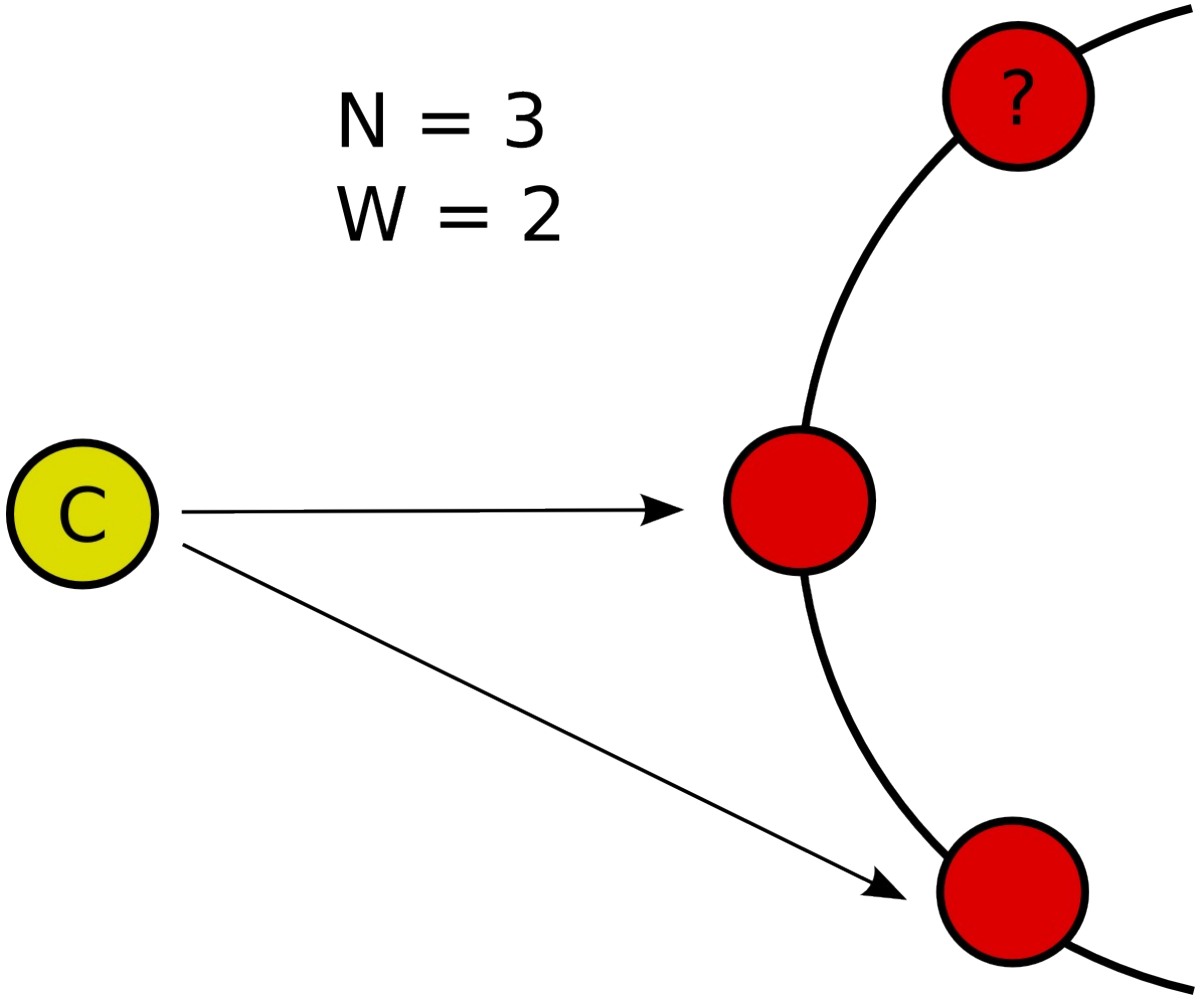
Choosing Consistency

Write	
Level	**Description**
ZERO	Hail Mary
ANY	1 replica (HH)
ONE	1 replica
QUORUM	(N / 2) +1
ALL	All replicas

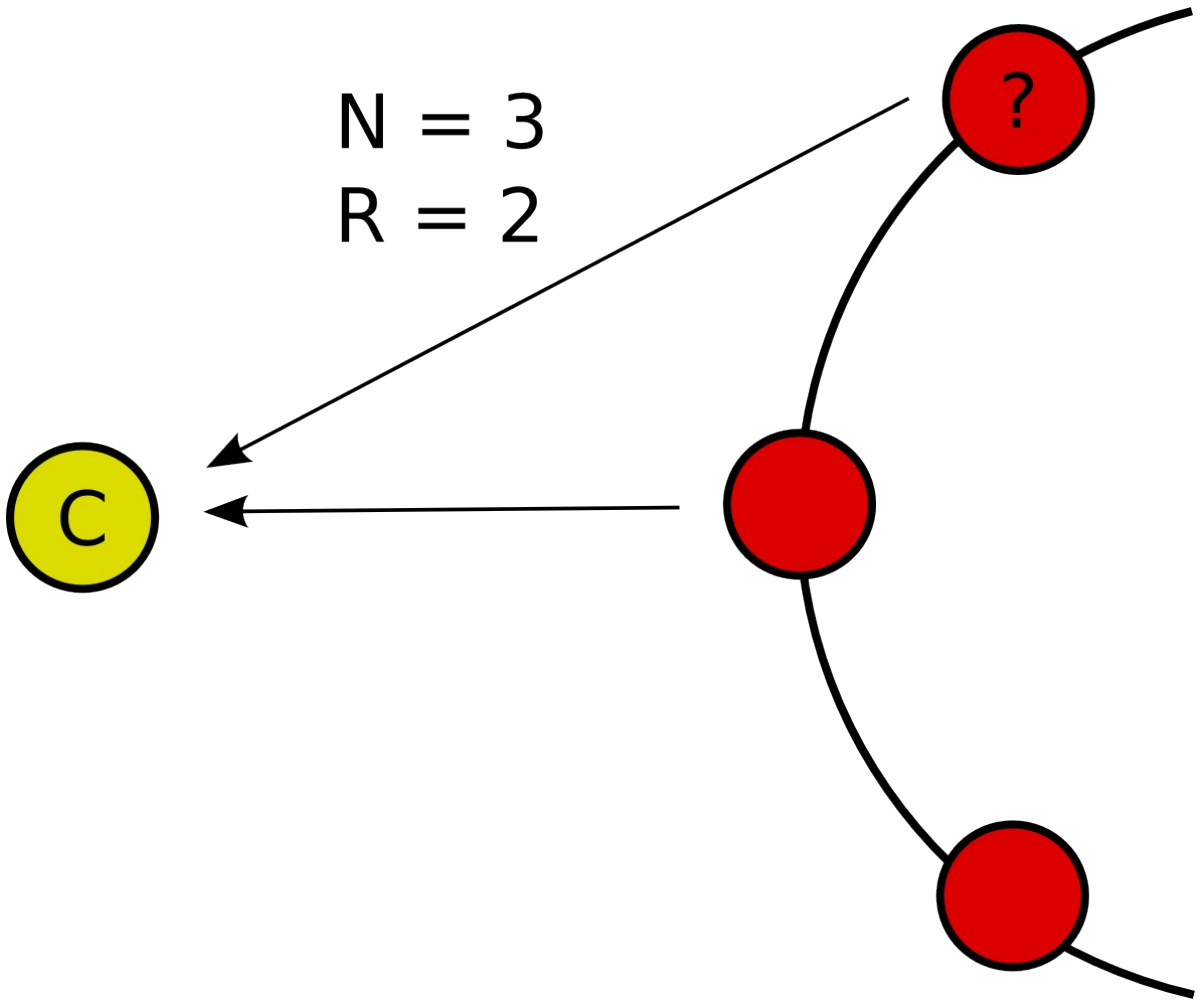
Read	
Level	**Description**
ZERO	N/A
ANY	N/A
ONE	1 replica
QUORUM	(N / 2) +1
ALL	All replicas

R + W > N

Quorum ((N/2) + 1)



Quorum ((N/2) + 1)



Data Model

Overview

- Keyspace
  - Uppermost namespace
  - Typically one per application
- ColumnFamily
  - Associates records of a similar kind
  - Record-level Atomicity
  - Indexed
- Column
  - Basic unit of storage

Sparse Table

Key 1	A=1	C=5	D=20	E=9
Key 2	B=19	D=22	E=7	
Key 3	A=3	B=4	F=9	H=7

Column

- `name`
  - `byte[]`
  - Queried against (predicates)
  - Determines sort order
- `value`
  - `byte[]`
  - Opaque to Cassandra
- `timestamp`
  - `long`
  - Conflict resolution (Last Write Wins)

Column Comparators

- Bytes
- UTF8
- TimeUUID
- Long
- LexicalUUID
- Composite (third-party)

http://github.com/edanuff/CassandraCompositeType

API

RPC

THRIFT

RPC

RPC

AVRO

RPC

Idiomatic Client Libraries

- Pelops, Hector (Java)
- Pycassa (Python)
- Cassandra (Ruby)
- Others …

http://wiki.apache.org/cassandra/ClientOptions

Code Samples

Pycassa – Python Client API

```
# creating a connection
from pycassa import connect
hosts = ['host1:9160', 'host2:9160']
client = connect('Keyspace1', hosts)

# creating a column family instance
from pycassa import ColumnFamily
cf = ColumnFamily(client, “Standard1”)

# reading/writing a column
cf.insert('key1', {'name': 'value'})
print cf.get('key1')['name']
```

1. http://github.com/vomjom/pycassa

Address Book – Setup

```
# conf/cassandra.yaml
keyspaces:
  - name: AddressBook
    column_families:
      - name: Addresses
        compare_with: BytesType
        rows_cached: 10000
        keys_cached: 50
        comment: 'No comment'
```

Adding an entry

```
key = uuid()

columns = {
    'first':  'Eric',
    'last':   'Evans',
    'email':  'eevans@rackspace.com',
    'city':   'San Antonio',
    'zip':    78250
}

addresses.insert(key, columns)
```

Fetching a record

```
# fetching the record by key
record = addresses.get(key)

# accessing columns by name
zipcode = record['zip']
city = record['city']
```

Indexing (manual)

```
# conf/cassandra.yaml
keyspaces:
  - name: AddressBook
    column_families:
      - name: Addresses
        compare_with: BytesType
        rows_cached: 10000
        keys_cached: 50
        comment: 'No comment'
      - name: ByCity
        compare_with: UTF8Type
```

Updating the index

```
key = uuid()

columns = {
    'first':  'Eric',
    'last':   'Evans',
    'email':  'eevans@rackspace.com',
    'city':   'San Antonio',
    'zip':    78250
}

addresses.insert(key, columns)
byCity.insert('San Antonio', {key: ''})
```

Indexing (auto)

```
# conf/cassandra.yaml
keyspaces:
  - name: AddressBook
    column_families:
      - name: Addresses
        compare_with: BytesType
        rows_cached: 10000
        keys_cached: 50
        comment: 'No comment'
        column_metadata:
          - name: city
            index_type: KEYS
```

Querying the Index

```
from pycassa.index import create_index_expression
from pycassa.index import create_index_clause

e = create_index_expression('city', 'San Antonio')
clause = create_index_clause([e])

results = address.get_indexed_slices(clause)

for (key, columns) in results.items():
    print “%(first)s %(last)s” % columns
```

Timeseries

```
# conf/cassandra.yaml
keyspaces:
  - name: Sites
    column_families:
      - name: Stats
        compare_with: LongType
```

Logging values

```
# time as long (milliseconds since epoch)
tstmp = long(time() * 1e6)

stats.insert('org.apache', {tstmp: value})
```

Slicing

```
begin = long(start * 1e6)

stats.get_range('org.apache',
                column_start=begin)

end = long((start + 86400) * 1e6)

stats.get_range(start='org.apache',
                finish='org.debian',
                column_start=begin,
                column_finish=end)
```

Questions?