ABOUT US

Mike Biglan, M.S.
- Twenty Ideas, Inc – twentyideas.com
- Analytic Spot – spo.td

Elijah Hamovitz
- code.org
- Analytic Spot – spo.td
A DATA MODEL MODELS DATA

• Framework to store and organize data
• Models things, their differences, and relationships between them
• Things can be real or virtual
IDEAL DATA MODEL PROPERTIES

• Easy to create
• Easy to interface with
• Quick, flexible querying
• Writing is direct and simple
• Easily understandable
• Scalable: can read, write, and store huge amount of data safely
SCALE AWAY, SCALE AWAY, SCALE AWAY

- Availability: fault tolerance, redundancy, supports multiple data centers
- Consistency: strong or tunable
- Huge amounts of data *(that won’t fit on single server)*
- High-speed of incoming and/or accessed data
COLUMNAR DATABASE

- Document-Store (e.g. MongoDB) and Columnar (e.g. Cassandra, HBase, Dynamo) are both “NoSQL”
- But modeling in Document-Store is quite different than Columnar
- Atomic unit of data storage
  - Document-Store: document
  - Relational Database: row
  - Columnar: column

CASSANDRA COLUMN:
- Name
- Value (or tombstone)
- Timestamp
- TTL
CASSANDRA

Highly Available Distributed Columnar Datastore that’s:

- Near-linearly scalable
- Fault tolerant, no master
- Tunable consistency
- Performant, especially for writes – *don’t read before write*
GETTING ON THE SCALE

Phase 1  Install Cassandra
Phase 2
Phase 3  Scale!
CQL != SQL
SQL/CQL STATEMENT FLOW

- SQL execution is complex
- CQL execution is relatively simple, hence tiny subset of syntax
- Much of CQL Query complexity is which node(s) to fetch/write/confirm data from/to
- So denormalize!
CQL != SQL

- CQL syntax is small subset of SQL

  ```
  SELECT <col1>, <col2>, ...
  FROM <table>
  LEFT JOIN <table2>...
  WHERE SEVERELY LIMITED
  GROUP BY <colx> HAVING ...
  ORDER BY SEVERELY LIMITED
  ```

mechanics.flite.com/blog/2013/11/05/breaking-down-the-cql-where-clause/
SO WHY THE LIMITATIONS?

Thinking of Cassandra as a relational database, it’s hard to understand:

• what is easy
• what is hard
• what is impossible
“Language serves not only to express thought but to make possible thoughts which could not exist without it.”

— Bertrand Russell
THE DISTORTION OF CQL

• Broken mental model hinders optimal modeling
• CQL falsely implies a relational data model and the design patterns that go with it
• To model Cassandra well, *know the underlying data structure*
DATA MODELING IN SQL
(NO SHARDING)

1. What are the Data?

2. What is the normalized data model?
   
   … months pass …

3. How are the data going to be queried?

4. Optimize any slow areas and/or bottlenecks
   
   • Add indexes, memcached/redis, sphinx/solr/elasticsearch, etc
1. What are the data?
2. What read-queries are needed?
3. How to denormalize during writes?
   - on initial write, or use external tools to make this sane

(Some) “premature” optimization is inherent and unavoidable
DATA ECOSYSTEM

To fully (and efficiently) enable everything SQL you are used to, must rely on the big(ish) data ecosystem:

• ElasticSearch, Solr, Sphinx
• Redis, Memcached
• Spark
• Spark Streaming or Storm (and Kafka)
COMPLEXITY OF INITIAL DATA MODEL

• Modeling with Relational DB
  • Items & their relationships

• Modeling with Cassandra
  • Items & their relationships
  • How/Where they are stored (sharding and hot spots)
  • What data we want to read
  • How (and how often) we write data into those models

CAN OPTIMIZE LATER
• Goal of a good black box is you can do a lot without knowing much about what’s inside

• CQL DOES NOT allow you to ignore what’s inside Cassandra
IN THE BEGINNING: THRIFT

- Around Cassandra version 0.8, Thrift started getting replaced with CQL.
- Thrift too low-level, but the interface had a close mapping to the underlying Cassandra data structure.
## THRIFT & CQL TERMINOLOGY

<table>
<thead>
<tr>
<th>THRIFT</th>
<th>CQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLUMN FAMILY</td>
<td>TABLE</td>
</tr>
<tr>
<td>ROW</td>
<td>PARTITION</td>
</tr>
<tr>
<td>COLUMN</td>
<td>CELL</td>
</tr>
<tr>
<td>[CELL NAME COMPONENT OR VALUE]</td>
<td>COLUMN</td>
</tr>
<tr>
<td>[GROUP OF CELLS WITH SHARED COMPONENT PREFIXES]</td>
<td>ROW</td>
</tr>
</tbody>
</table>

CREATE TABLE employees (
  company text,
  name text,
  age int,
  role text,
  PRIMARY KEY ((company), name)
);
CREATE TABLE employees (  
  company text,  
  name text,  
  age int,  
  role text,  
  PRIMARY KEY  
  ((company), name)  
);

> SELECT * from employees;

<table>
<thead>
<tr>
<th>company</th>
<th>name</th>
<th>age</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo, inc</td>
<td>Fred</td>
<td>31</td>
<td>coder</td>
</tr>
<tr>
<td>Foo, inc</td>
<td>Sara</td>
<td>39</td>
<td>boss</td>
</tr>
<tr>
<td>BarCo</td>
<td>Bill</td>
<td>50</td>
<td>SQL guru</td>
</tr>
<tr>
<td>BarCo</td>
<td>Jane</td>
<td>20</td>
<td>hotshot</td>
</tr>
</tbody>
</table>
CREATE TABLE employees (  
  company text,  
  name text,  
  age int,  
  role text,  
  PRIMARY KEY  
  ((company), name)  
);  

employees = {  
  "Foo, inc" : {  
    "Fred:age" : 31,  
    "Fred:role" : "coder",  
    "Sara:age" : 39,  
    "Sara:role" : "boss"  
  },  
  "BarCo" : {  
    "Bill:age" : 50,  
    "Bill:role" : "SQL guru"  
  }  
}
WIDE PARTITIONS
(FORMERLY WIDE “ROWS”)

• Based on Thrift “rows”, so actually wide “partitions”

• Columns are the clustering key values with the column-name suffix

• Up to 2 billion (but don’t do this)
SETS, MAPS, AND LISTS: OH MY

- Sets/Maps/List still column-level storage
- Enabling Schemaless, but can result in long column names

www.slideshare.net/DataStax/understanding-how-cql3-maps-to-cassandras-internal-data-structure (p 52-71)
PARTITION AND CLUSTERING KEY

CQL Where-Clause variants

- None (kind of)
- key1 & key2
- key1 & key2 & key3
- key1 & key2 & key3 & key4
CREATE TABLE breaks (  
  company text,  
  year int,  
  month int,  
  day int,  
  employee text,  
  reason text,  
  PRIMARY KEY  
  ((company, year),  
   month, day, employee))  
);
CREATE TABLE breaks (  
  company text,  
  year int,  
  month int,  
  day int,  
  employee text,  
  reason text,  
  PRIMARY KEY  
  ((company, year),  
   month, day, employee)  
);
THEN WHAT IS EASY?

With a dictionary of ordered dictionaries:

- Grabbing the data (or subset) from a partition-key
- Getting a slice of data (*uses linear search*) based on a partition-key

```python
data = {
    "Foo, Inc:2014": {
        "11": {
            "27": {
                "Fred:reason": "Thanksgiving"
            }
        }
    },
    "12": {
        "25": {
            "Fred:reason": "Christmas",
            "Sara:reason": "Christmas"
        }
    },
    "26": {
        "Sara:reason": "Boxing day"
    }
}
```
WHAT IS HARD?
(I.E. COMMON SQL PATTERNS)

• Unique and Group by
• Ordered
• Inverted Index
GROUP-BY & COUNTERS

- Often group-by is used for counting
- Use counter columns or other tools (e.g. elasticsearch)

```sql
CREATE TABLE employee_break_counts (  
  company text,  
  employee text,  
  break_counts counter,  
  PRIMARY KEY  
  ((company), employee)  
);
```
ORDERING OR INVERTED-INDEX

- Redundant table, but ordered by new "column"
- Depending on needs, this can store the order-field and lookup key OR some/all of the other data in that table
- If a read will generate more than a few subsequent child reads then some/all the other data should be included

CREATE TABLE employees_by_age (
  company text,
  id int,
  age int,
  name text,
  role text,
  PRIMARY KEY ((company), age, id)
);
### C* Modeling Anti-Patterns

<table>
<thead>
<tr>
<th>C* Guidelines</th>
<th>SQL Guidelines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes are cheap/fast</td>
<td>Minimize writes</td>
</tr>
<tr>
<td>Storage is cheap</td>
<td>Minimize duplication of data</td>
</tr>
<tr>
<td>Partitions are inherent</td>
<td>Shard at your own risk</td>
</tr>
<tr>
<td>Strict composite keys</td>
<td>Flexible secondary indexes</td>
</tr>
<tr>
<td>Simple queries</td>
<td>Complex queries</td>
</tr>
</tbody>
</table>
## C* Modeling Patterns

<table>
<thead>
<tr>
<th>C* Guidelines</th>
<th>C* Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes are cheap/quick</td>
<td>Duplicate your data</td>
</tr>
<tr>
<td>Storage is cheap</td>
<td></td>
</tr>
<tr>
<td>Partitions are inherent</td>
<td>Avoid hot spots</td>
</tr>
<tr>
<td>Strict composite keys</td>
<td>Design tables around queries</td>
</tr>
<tr>
<td>Simple queries</td>
<td></td>
</tr>
</tbody>
</table>
Questions?

Mike Biglan
mike@twentyideas.com
@twentyideas

Elijah Hamovitz
elijah@code.org