

Cassandra Indexing Techniques

Ed Anuff

Founder, Usergrid

Cassandra Summit SF July, 2011

usergrid_

Agenda

- Background
- Basics of Indexes
- Native Secondary Indexes
- "Wide rows" and CF-based Indexes
- Inverted-indexes Using SuperColumns
- Inverted-indexes Using Composite Columns
- Q&A

Background

This presentation is based on:

- What we learned over the last year in building a highly indexed system on Cassandra
- Participation in the Hector client project
- Common questions and issues posted to the Cassandra mailing lists

Brief History - Cassandra 0.6

- No built-in secondary indexes
- All indexes were custom-built, usually using super-columns
- Pros
 - Forced you to learn how Cassandra works
- Cons
 - Lots of work
 - Super-columns proved a dead-end

Brief History - Cassandra 0.7

- Built-in secondary indexes
- New users flocked to these
- Pros
 - Easy to use, out of the box
- Cons
 - Deceptively similar to SQL indexes but not the same
 - Reinforce data modeling that plays against Cassandra's strengths

Present Day

- New users can now get started with Cassandra without really understanding it (CQL, etc.)
- Veteran users are using advanced techniques (Composites, etc.) that aren't really documented anywhere*
- New user panic mode when they try to go to the next level and suddenly find themselves in the deep end

*Actually, they are, Google is your friend...

usergrid_

Let's try to bridge the gap...

usergrid_

A Quick Review

There are essentially two ways of finding rows:

The Primary Index
(row keys)

Alternate Indexes
(everything else)

usergrid_

The “Primary Index”

- The “primary index” is your row key*
- Sometimes it’s meaningful (“natural id”):

```
Users = {  
  "edanuff" : {  
    email: "ed@anuff.com"  
  }  
}
```

*Yeah. No. But if it helps, yes.

usergrid_

The “Primary Index”

- The “primary index” is your row key*
- But usually it’s not:

```
Users = {  
  "4e3c0423-aa84-11e0-a743-58b0356a4c0a" : {  
    username: "edanuff",  
    email: "ed@anuff.com"  
  }  
}
```

*Yeah. No. But if it helps, yes.

usergrid_

Get vs. Find

- Using the row key is the best way to retrieve something if you've got a precise and immutable 1:1 mapping
- If you find yourself ever planning to iterate over keys, you're probably doing something wrong
 - i.e. avoid the Order Preserving Partitioner*
- Use alternate indexes to find (search) things

*Feel free to disregard, but don't complain later

usergrid_

Alternate Indexes

Anything other than using the row key:

- Native secondary indexes
- Wide rows as lookup and grouping tables
- Custom secondary indexes

Remember, there is no magic here...

usergrid_

Native Secondary Indexes

- Easy to use
- Look (deceptively) like SQL indexes, especially when used with CQL

```
CREATE INDEX ON Users (username) ;
```

```
SELECT * FROM Users WHERE  
username="edanuff";
```

usergrid_

Under The Hood

- Every index is stored as its own "hidden" CF
- Nodes index the rows they store
- When you issue a query, it gets sent to all nodes
- Currently does equality operations, the range operations get performed by in memory by coordinator node

Some Limitations

- Not recommended for high cardinality values (i.e. timestamps, birthdates, keywords, etc.)
- Requires at least one equality comparison in a query – not great for less-than/greater-than/range queries
- Unsorted - results are in token order, not query value order
- Limited to search on data types Cassandra natively understands

I can't live with those limitations,
what are my options?



Complain on the mailing list



Switch to Mongo



Build my indexes in my application

That sounds hard...

Actually, it's not, but it does require
us to revisit some of Cassandra's
unique features

usergrid_

Wide Rows

“Why would a row need 2B columns”?

- Basis of all indexing, organizing, and relationships in Cassandra
- If your data model has no rows with over a hundred columns, you're either doing something wrong or shouldn't be using Cassandra*

*IMHO 😊

usergrid_

Conventional Rows As Records

```
Users = {  
  "4e3c0423-..." : {  
    username: "edanuff",  
    email: "ed@anuff.com",  
    ... : ...  
  },  
  "e5d61f2b-..." : {  
    username: "jdoe",  
    email: "john.doe@gmail.com",  
    ... : ...  
  }  
}
```

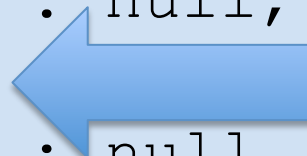
usergrid_

Wide Row For Grouping

```
Departments = {
```

```
  "Engineering" : {  
    "e5d61f2b-..." : null,  
    "e5d61f2b-..." : null,  
    "e66afd40-..." : null,  
    "e719962b-..." : null,  
    "e78ece0f-..." : null,  
    "e80a17ba-..." : null,  
    ... : ...,  
  }
```

Column names
are keys of rows
in "Users" CF



```
}
```


usergrid_

Wide Row As A Simple Index

Comparator = "UTF8Type"

Indexes = {

"User_Keys_By_Last_Name" : {



```
"adams"      : "e5d61f2b-...",  
"alden"     : "e80a17ba-...",  
"anderson"  : "e5d61f2b-...",  
"davis"     : "e719962b-...",  
"doe"       : "e78ece0f-...",  
"franks"    : "e66afd40-...",  
... : ...,
```

}

}

usergrid_

Column Families As Indexes

- CF column operations very fast
- Column slices can be retrieved by range, are always sorted, can be reversed, etc.
- If target key a TimeUUID you get both grouping and sort by timestamp
 - Good for inboxes, feeds, logs, etc. (Twissandra)
- Best option when you need to combine groups, sort, and search
 - Search friends list, inbox, etc.

But, only works for 1:1

What happens when I've got one to many?

```
Indexes = {  
  "User_Keys_By_Last_Name" : {  
    "adams"      : "e5d61f2b-...",  
    "alden"      : "e80a17ba-...",  
    "anderson"   : "e5d61f2b-...",  
    "anderson"   : "e719962b-...",  
    "doe"        : "e78ece0f-...",  
    "franks"     : "e66afd40-...",  
    ... : ...,  
  }  
}
```



Not Allowed

usergrid_

SuperColumns to the rescue?

```
Indexes = {
```

```
  "User_Keys_By_Last_Name" : {  
    "adams" : {  
      "e5d61f2b-..." : null  
    },  
    "anderson" : {  
      "e5d61f2b-..." : null,  
      "e66afd40-..." : null  
    }  
  }  
}
```

```
}
```

usergrid_

Use with caution

- Not officially deprecated, but not highly recommended either
- Sorts only on the supercolumn, not subcolumn
- Some performance issues
- What if I want more nesting?
 - Can subcolumns have subcolumns? NO!
- Anecdotally, many projects have moved away from using supercolumns

So, let's revisit regular CF's

What happens when I've got one to many?

```
Indexes = {  
  "User_Keys_By_Last_Name" : {  
    "adams"      : "e5d61f2b-...",  
    "alden"     : "e80a17ba-...",  
    "anderson"  : "e5d61f2b-...",  
    "anderson"  : "e719962b-...",  
    "doe"       : "e78ece0f-...",  
    "franks"    : "e66afd40-...",  
    ... : ...,  
  }  
}
```



Not Allowed

usergrid_

So, let's revisit regular CF's

What if we could turn it back to one to one?

```
Indexes = {  
  "User_Keys_By_Last_Name" : {  
    {"adams", 1}      : "e5d...",  
    {"alden", 1}     : "e80...",  
    {"anderson", 1}  : "e5f...",  
    {"anderson", 2}  : "e71...",  
    {"doe", 1}       : "e78...",  
    {"franks", 1}    : "e66...",  
    ... : ...,  
  }  
}
```



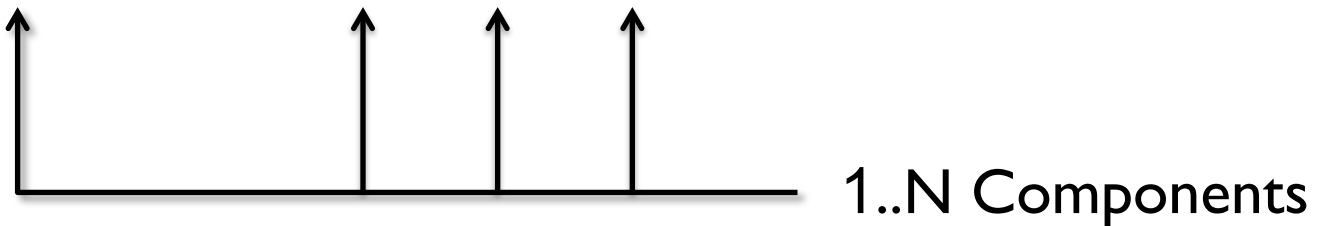
Allowed!!!

usergrid_

Composite Column Names

Comparator = "CompositeType" or "DynamicCompositeType"

{ "anderson", 1, 1, 1, ... } : "e5f..."



1..N Components

Build your column name out of one or more
“component” values which can be of any of the
columns types Cassandra supports
(i.e. UTF8, IntegerType, UUIDType, etc.)

Composite Column Names

Comparator = "CompositeType" or "DynamicCompositeType"



```
{ "anderson", 1, 1, 1, ... } : "e5f..."  
{ "anderson", 1, 1, 2, ... } : "e5f..."  
{ "anderson", 1, 2, 1, ... } : "e5f..."  
{ "anderson", 1, 2, 2, ... } : "e5f..."
```

Sorts by component values using each
component type's sort order

Retrieve using normal column slice technique

usergrid_

Two Types Of Composites

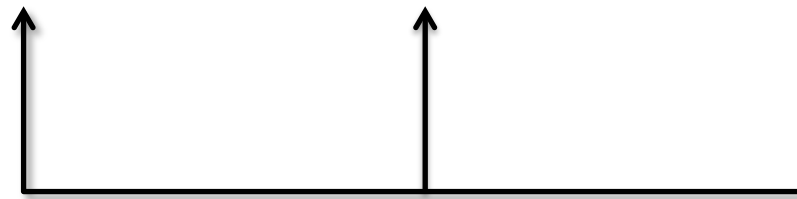
- column_families:
 - name: My_Composite_Index_CF
 - compare_with:
CompositeType(UTF8Type, UUIDType)
- name: My_Dynamic_Composite_Index_CF
- compare_with:
DynamicCompositeType(s=>UTF8Type, u=>UUIDType)

Main difference for use in indexes is whether you need to create one CF per index vs one CF for all indexes with one row per index

Static Composites

- column_families:
 - name: My_Composite_Index_CF
 - compare_with:
CompositeType(UTF8Type, UUIDType)

```
{"anuff", "e5f..."} : "e5f..."
```



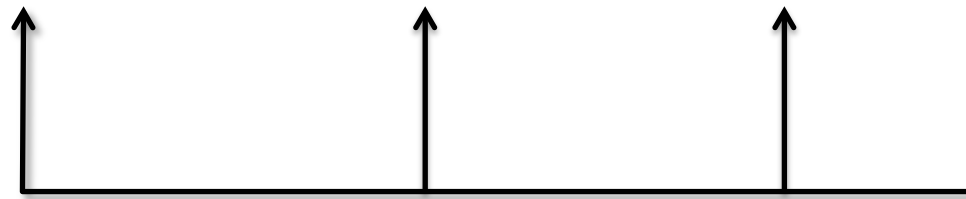
Fixed # and order
defined in column
configuration

usergrid_

Dynamic Composites

- column_families:
 - name: My_Dynamic_Composite_Index_CF
 - compare_with:
DynamicCompositeType (s=>UTF8Type, u=>UUIDType)

`{"anuff", "e5f...", "e5f..."}` : `"..."`



Any # and
order of
types at
runtime

Typical Composite Index Entry

```
{<term 1>, ..., <term N>, <key>, <ts>}
```

<term 1...N> - terms to query on (i.e. last_name, first_name)

<key> - target row key

<ts> - unique timestamp, usually time-based UUID

usergrid_

How does this work?

- Queries are easy
 - regular column slice operations
- Updates are harder
 - Need to remove old value and insert the new value
 - Uh oh, read before write??!!!

Example – Users By Location

- We need 3 Column Families (not 2)
- First 2 CF's are obvious:
 - Users
 - Indexes
- We also need a third CF:
 - Users_Index_Entries

usergrid_

Users CF

Comparator = "BytesType"

```
Users = {  
  <user_key> : {  
    "username" : "...",  
    "location" : <location>,  
    ... : ...,  
  }  
}
```

usergrid_

Indexes CF

Comparator = "CompositeType"

```
Indexes = {  
  "Users_By_Location" : {  
    {<location>, <user_key>, <ts>} : ...,  
    ... : ...,  
  }  
}
```

usergrid_

Users Index Entries CF

Comparator = "CompositeType"

```
Users_Index_Entries = {  
  <user_key> : {  
    {"location", <ts 1>} : <location 1>,  
    {"location", <ts 2>} : <location 2>,  
    {"location", <ts N>} : <location N>,  
    {"last_name", <ts 1>} : "...",  
    ... : .../  
  }  
}
```

usergrid_

Users Index Entries CF

Comparator = "CompositeType"

```
Users_Index_Entries = {  
  <user_key> : {  
    {"location", <ts 1>} : <location 1>,  
    {"location", <ts 2>} : <location 2>,  
    {"location", <ts N>} : <location N>,  
    {"last_name", <ts 1>} : "...",  
    ... : ...,  
  }  
}
```

usergrid_

Users Index Entries CF

Comparator = "CompositeType"

```
Users_Index_Entries = {  
  <user_key> : {  
    {"location", <ts 1>} : <location 1>,  
    {"location", <ts 2>} : <location 2>,  
    {"location", <ts N>} : <location N>,  
    {"last_name", <ts 1>} : "...",  
    ... : .../  
  }  
}
```

usergrid_

Updating The Index

- We read previous index values from the Users_Index_Entries CF rather than the Users CF to deal with concurrency
- Columns in Index CF and Users_Index_Entries CF are timestamped so no locking is needed for concurrent updates

Get Old Values For Column

```
SELECT {"location"}..{"location",*}  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
BEGIN BATCH
```

```
DELETE {"location", ts1}, {"location", ts2}, ...  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
DELETE {<value1>, <user_key>, ts1}, {<value2>, <user_key>, ts2}, ...  
FROM Users_By_Location WHERE KEY = <user_key>;
```

```
UPDATE Users_Index_Entries SET {"location", ts3} = <value3>  
WHERE KEY = <user_key>;
```

```
UPDATE Indexes SET {<value3>, <user_key>, ts3} = null  
WHERE KEY = "Users_By_Location";
```

```
UPDATE Users SET location = <value3>  
WHERE KEY = <user_key>;
```

```
APPLY BATCH
```

usergrid_

Remove Old Column Values

```
SELECT {"location"}..{"location",*}  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
BEGIN BATCH
```

```
DELETE {"location", ts1}, {"location", ts2}, ...  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
DELETE {<value1>, <user_key>, ts1}, {<value2>, <user_key>, ts2}, ...  
FROM Users_By_Location WHERE KEY = <user_key>;
```

```
UPDATE Users_Index_Entries SET {"location", ts3} = <value3>  
WHERE KEY = <user_key>;
```

```
UPDATE Indexes SET {<value3>, <user_key>, ts3} = null  
WHERE KEY = "Users_By_Location";
```

```
UPDATE Users SET location = <value3>  
WHERE KEY = <user_key>;
```

```
APPLY BATCH
```

usergrid_

Insert New Column Values In Index

```
SELECT {"location"}..{"location",*}  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
BEGIN BATCH
```

```
DELETE {"location", ts1}, {"location", ts2}, ...  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
DELETE {<value1>, <user_key>, ts1}, {<value2>, <user_key>, ts2}, ...  
FROM Users_By_Location WHERE KEY = <user_key>;
```

```
UPDATE Users_Index_Entries SET {"location", ts3} = <value3>  
WHERE KEY = <user_key>;
```

```
UPDATE Indexes SET {<value3>, <user_key>, ts3) = null  
WHERE KEY = "Users_By_Location";
```

```
UPDATE Users SET location = <value3>  
WHERE KEY = <user_key>;
```

```
APPLY BATCH
```

usergrid_

Set New Value For User

```
SELECT {"location"}..{"location",*}  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
BEGIN BATCH
```

```
DELETE {"location", ts1}, {"location", ts2}, ...  
FROM Users_Index_Entries WHERE KEY = <user_key>;
```

```
DELETE {<value1>, <user_key>, ts1}, {<value2>, <user_key>, ts2}, ...  
FROM Users_By_Location WHERE KEY = <user_key>;
```

```
UPDATE Users_Index_Entries SET {"location", ts3} = <value3>  
WHERE KEY = <user_key>;
```

```
UPDATE Indexes SET {<value3>, <user_key>, ts3} = null  
WHERE KEY = "Users_By_Location";
```

```
UPDATE Users SET location = <value3>  
WHERE KEY = <user_key>;
```

```
APPLY BATCH
```

usergrid_

Frequently Asked Questions

- Do I need locking for concurrency?
 - No, the index will always be eventually consistent
- What if something goes wrong?
 - You will have to have provisions to repeat the batch operation until it completes, but its idempotent, so it's ok
- Can't I get a false positive?
 - Depending on updates being in-flight, you might get a false positive, if this is a problem, filter on read
- Who else is using this approach?
 - Actually very common with lots of variations, this isn't the only way to do this but at least composite format is now standard

Some Things To Think About

- Indexes can be derived from column values
 - Create a “last_name, first_name” index from a “fullname” column
 - Unroll a JSON object to construct deep indexes of serialized JSON structures
- Include additional denormalized values in the index for faster lookups
- Use composites for column values too not just column names

How can I learn more?

Sample implementation using Hector:

<https://github.com/edanuff/CassandraIndexedCollections>

JPA implementation using this for Hector:

<https://github.com/riptano/hector-jpa>

Jira entry on native composites for Cassandra:

<https://issues.apache.org/jira/browse/CASSANDRA-2231>

Background blog posts:

<http://www.anuff.com/2011/02/indexing-in-cassandra.html>

<http://www.anuff.com/2010/07/secondary-indexes-in-cassandra.html>

usergrid_

What is Usergrid?

- Cloud PaaS backend for mobile and rich-client applications
- Powered by Cassandra
- In private Beta now
- Entire stack to be open-sourced in August so people can run their own
- Sign up to get admitted to Beta and to be notified when about source code

<http://www.usergrid.com>

usergrid_

Thank You

usergrid_