



DataStax Hands-On Modelling

Rob Murphy & Aaron Regis

14th June 2017

Agenda

1	Storing data in DSE C*
2	Data Modelling, CQL basics
3	Hands-On Primary Keys

Storing data in DSE C*

Write request

user_id : 4
name : Gregg

user_id : 4
name : Gregg

MEMTABLE

FLUSH

SSTABLE

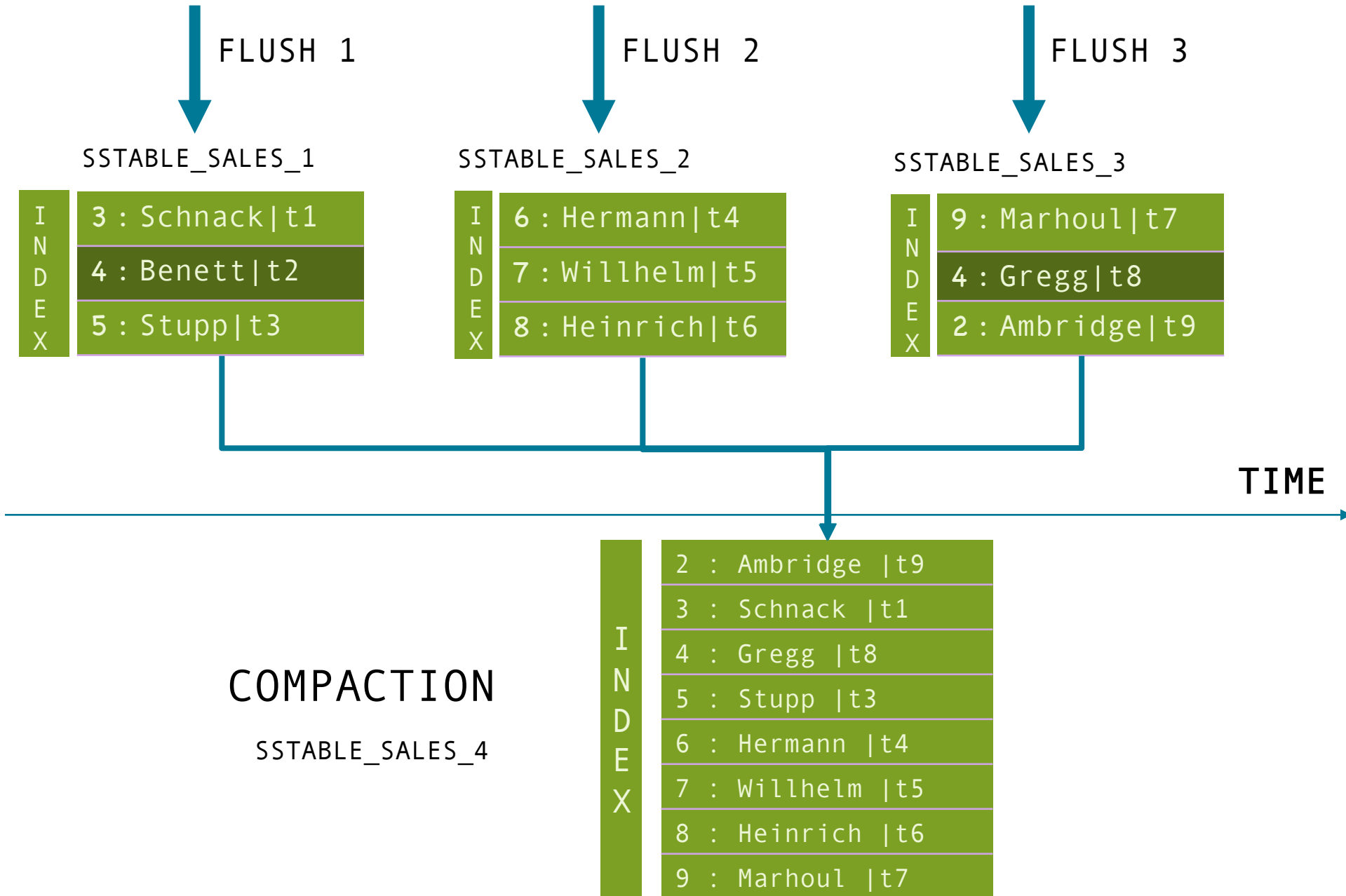
I
N
D
E
X

3 : Schnack
4 : Gregg
6 : Stupp
8 : Ambridge

IMMUTABLE
ORDERD

COMMIT LOG

user_id : 4
name : Gregg



Data Modeling Objectives

Data modeling objectives

1. Get your data **out of DSE**
2. Reduce query **latency**, make your queries **faster**
3. Avoid **disaster** in production

Data modeling methodology

Design by query

- first, know your **functional queries**
- design the **table(s)** for direct access
- just **denormalize** if necessary
- Spread data evenly around the cluster
- Minimize the number of partitions read

Output of design phase = **schema.cql**

Then start coding

Know your functional queries

Query:

`find users by id group by region and ordered by join date`

- Grouping by an attribute
- Ordering by an attribute
- Filtering based on some set of conditions
- Enforcing uniqueness in the result set

The partition key

Role

Partition key

- main entry point for query (INSERT/SELECT ...)
- help distribute/locate data on the cluster

No partition key = full cluster scan

How to choose correct partition key ?

Good partition column

- choose **functional** identifier
- high cardinality (lots of **distinct values**)

Query:

Find all sales by sales representative?

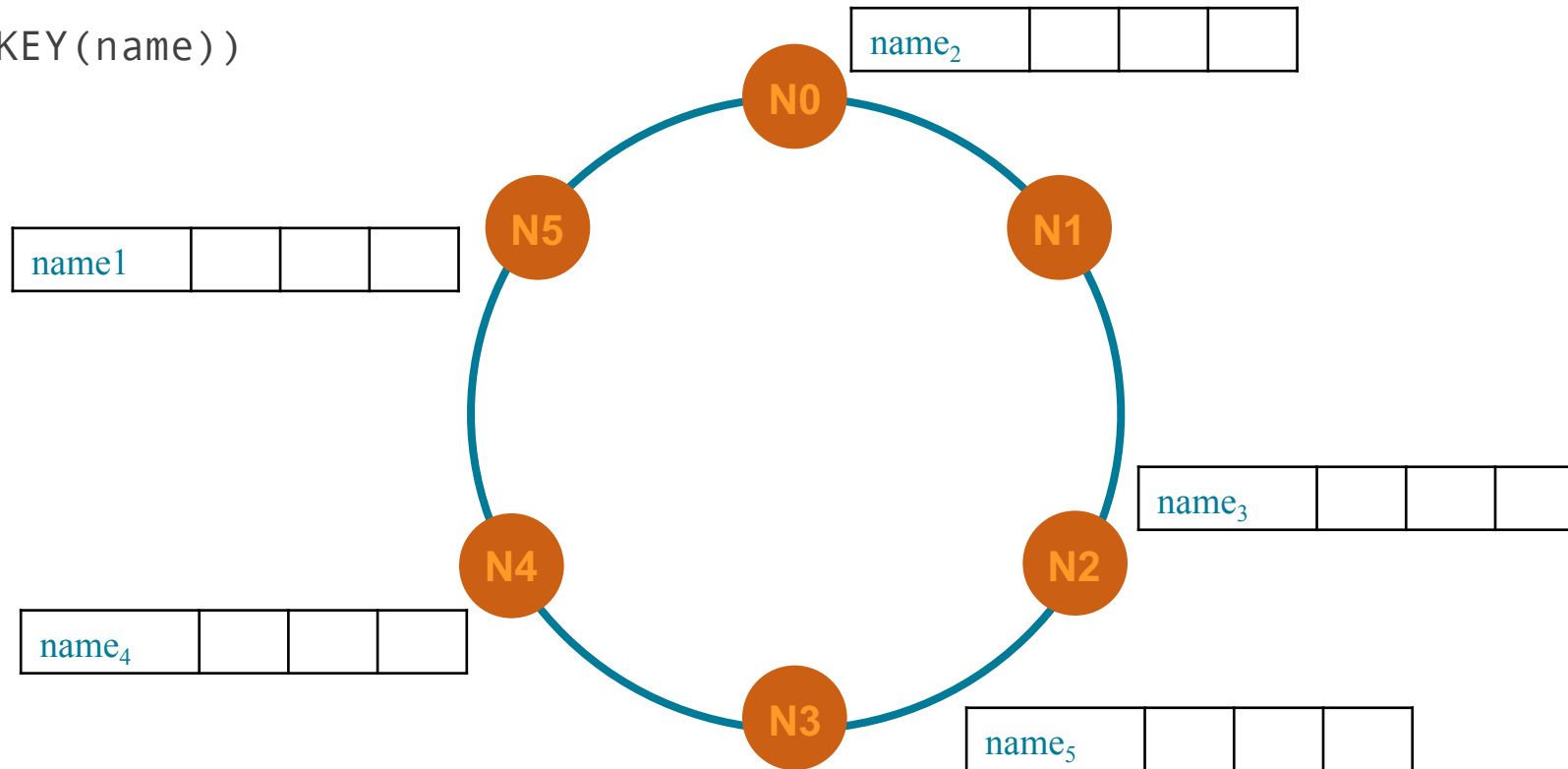
```
CREATE TABLE sales_by_repname (  
  name text,  
  sdate tuuid,  
  item text,  
  price double,  
  ...,  
  PRIMARY KEY(name));
```

partition key (#partition)



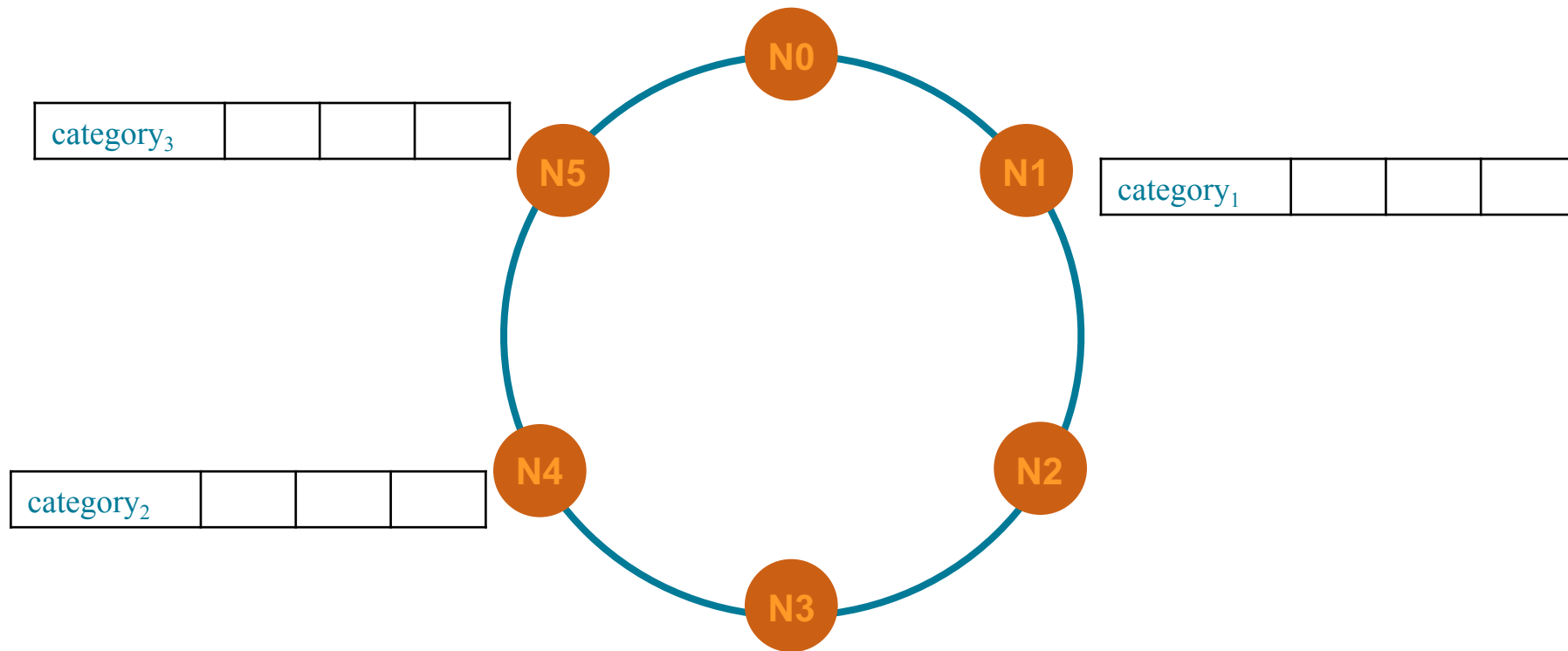
Example of good partition key

```
CREATE TABLE sales_by_repname(  
  name text,  
  ... ,  
  PRIMARY KEY(name))
```



Example of bad partition key

```
CREATE TABLE sales_by_cat(  
  category text,  
  ...,  
  PRIMARY KEY(category))
```



CRUD operations

```
INSERT INTO sales(name, item, price) VALUES('Gregg', 'iPhone7', 799);
```

```
UPDATE sales SET qty = 1 WHERE name = Gregg;
```

```
DELETE qty FROM sales WHERE name = Gregg;
```

```
SELECT rev FROM sales WHERE name = Gregg;
```

No Clustering Columns

PRIMARY KEY (name)

// no duplicate primary keys

name	dt	item	qty	price	rev
Gregg	20160101	PlayStation4	1	500	500
Schnack	20160102	IPhone 7	2	799	1598
Schnack	20160103	IPhone 7	1	799	799

```
SELECT * FROM sales_by_repname WHERE name = 'Schnack';
```


Composite partition key

Multiple columns for partition key

- always **known in advance** (INSERT/SELECT ...)
- are **hashed together** to the same token value

```
CREATE TABLE sales_by_name_and_date(  
    name  text,  
    dt    date,  
    item  text,  
    ...,  
    PRIMARY KEY((name,sdate))
```

```
SELECT * FROM sales WHERE name = ... AND dt = ...
```

```
SELECT * FROM sales WHERE name = ... AND dt IN (xxx, yyy ...)
```

Compound Partition Key

PRIMARY KEY ((name,dt))

// hash(name,dt) → token

name	dt	item	qty	price	rev
Gregg	20160101	PlayStation4	1	500	500
Schnack	20160102	IPhone 7	2	799	1598
Schnack	20160103	IPhone 7	1	799	799

```
SELECT * FROM sales WHERE name = 'Schnack' AND date = '1/2/2016';
```

The clustering column(s)

Role

Clustering column(s)

- simulate 1 – N relationship
- and sort data (logically & on disk)

Clustered table (1 – N)

```
CREATE TABLE sales(  
  name  text,  
  dt    date,  
  item  text,  
  qty   int,  
  rev   int,  
  PRIMARY KEY((name), dt));
```

partition key

clustering column
(sorted)

uniqueness

Recommended syntax

```
PRIMARY KEY((sensor_id), date))
```

Clustering Columns Create Wide Rows

PRIMARY KEY (name,dt)

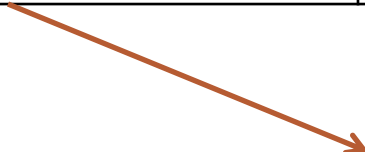
// default sort and range queries

name	dt		
Schnack	20160102		
	iPhone7, 2, 1598		
Gregg	20160101	20160102	20160103
	iPhone7, 2, 1598	iPhone6, 2, 1899	SonyPlaystation 4, 1, 399

```
SELECT * FROM sales WHERE name = 'Gregg' and dt > '1/1/2016';
```

What's Stored With Each Column?

name	dt		
Schnack	20160102		
	'Microsoft Xbox', 1, 299.00		
Gregg	20160101	20160102	20160103
	'Sony Playstation 4', 1, 399.00	'Apple Watch', 1, 499.00	'Mac Book Pro', 1, 2300.00



column name : "item"
column value : "Sony Playstation 4"
timestamp : 1353890782373000
TTL : 3600

Columns relationship and ordering

```
CREATE TABLE sales (  
  name  text,  
  dt    date,  
  item  text,  
  qty   integer,  
  price double,  
  rev   double,  
  PRIMARY KEY((name), dt))  
WITH CLUSTERING ORDER BY (dt DESC)
```

Diagram illustrating column relationships and ordering:


- A bracket groups `name` and `dt`, with an arrow pointing to `name (1) <-----> (N) dt`.
- A bracket groups `item`, `qty`, `price`, and `rev`, with an arrow pointing to `dt (1) <-----> (1) (item, qty, price, rev)`.

name	dt		
Schnack	20160102		
	iPhone7, 2, 1598		
Gregg	20160101	20160102	20160103
	iPhone7, 2, 1598	iPhone6, 2, 1899	SonyPlaystation 4, 1, 399

```
SELECT * FROM sales WHERE name = 'Gregg' and dt > '1/1/2016';
```


Multiple clustering columns

```
CREATE TABLE sales (  
  name  text,  
  cat   text,  
  dt    date,  
  item  text,  
  qty   integer,  
  price double,  
  rev   double,  
  PRIMARY KEY((name), cat, dt))  
  WITH CLUSTERING ORDER BY (cat ASC, dt DESC)
```



name	cat,dt
Schnack	Apple
	20160102
	iPhone7, 2, 1598
Gregg	Apple
	20160101
	iPhone7, 2, 1598

```
SELECT * FROM sales  
  WHERE name = 'Gregg' AND  
        cat = 'Apple' AND dt >= 1/2/16 AND dt <= 1/3/16;
```

Primary key summary

PRIMARY KEY((name), cat, dt))



Provides uniqueness

Primary key summary

PRIMARY KEY((name), cat, dt))




Used to locate **node** in the cluster

Used to locate **partition** in the node

Primary key summary

PRIMARY KEY((name), cat, dt))



Used to lookup rows in a partition

Used for data sorting and range queries

Other critical details

Huge partitions

```
PRIMARY KEY((sensor_id), dt))
```

Data for the same sensor stay in the same partition on disk

Huge partitions

PRIMARY KEY((sensor_id), dt))

Data for the **same sensor** stay in the **same partition** on disk

If insert rate = 100/sec, how big is my partition after 1 year ?

→ $100 \times 3600 \times 24 \times 365 = 3\,153\,600\,000$ cells on disks

Huge partitions

PRIMARY KEY((sensor_id), dt))

Theoretical limit of # cells for a partition = 2×10^9

Practical limit for a partition on disk

- 100Mb
- 100 000 – 1 000 000 cells

Reasons ? Make maintenance operations easier

- compaction
- repair
- bootstrap ...

Sub-partitioning techniques

PRIMARY KEY((sensor_id, day), dt))

→ 100 x 3600 x 24 = 8 640 000 cells on disks ✓

Sub-partitioning techniques

PRIMARY KEY((sensor_id, day), dt))

→ $100 \times 3600 \times 24 = 8\,640\,000$ cells on disks ✓

But impact on queries:

- need to provide sensor_id & day for any query
- how to fetch data across N days ?

Data deletion and tombstones

```
DELETE FROM sensor_data  
WHERE sensor_id = .. AND dt = ...
```

Logical deletion of data but:

- new physical "tombstone" column on disk
- disk space usage will increase !

The "tombstone" columns will be purged later by compaction process ...

Lab 3 : Hands-on Primary Keys

Thank You!