The Cassandra Query Language

CS185C: Introduction to NoSQL Databases

Suneuy Kim

References

- Cassandra: The Definitive Guide, 2nd Edition Distributed Data at Web Scale By Jeff Carpenter, Eben Hewitt, June 2016
- Cassandra Query Language, https://cassandra.apache.org/doc/cql3/CQL.html

Cassandra Data Structures

- Column: a name and value pair
- Row: A container for columns and it is referenced by a primary key (also called row key)
- Table: A container for rows
  - Logical division that associates similar data.
  - A user table, A hotel table, an address book table, etc.
- Keyspace: A container for tables
- Cluster: A container for keyspaces that spans one or more nodes.

Primary Key

- Primary key: unique identifier of each row

```
CREATE TABLE crossfit_gyms (
    gym_name text,
    city text,
    state_province text,
    country_code text,
    PRIMARY KEY (gym_name)
);
```

http://datascale.io/cassandra-partitioning-and-clustering-keys-explained/

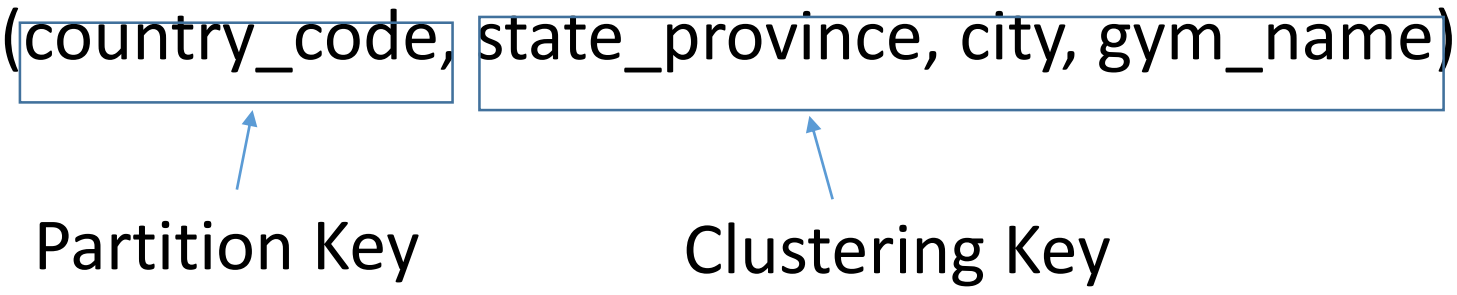
Partition Key and Clustering Key

- Partition key: responsible for the distribution of data among the nodes
  - When a primary key consists of a single column, it serves as the partition key as well.
  - When a primary key consists of multiple columns, the first column of the primary key serves as the partition key.
- Clustering Keys
  - Except for the first column, each additional column that is added to the Primary Key clause is called a Clustering Key.
  - A clustering key is responsible for sorting data within the partition.
  - By default, the clustering key columns are sorted in ascending order.

http://datascale.io/cassandra-partitioning-and-clustering-keys-explained/

Partition Key and Clustering Key

```
CREATE TABLE crossfit_gyms_by_location (
    country_code text,
    state_province text,
    city text,
    gym_name text,
    PRIMARY KEY (country_code, state_province, city, gym_name)
);
```



http://datascale.io/cassandra-partitioning-and-clustering-keys-explained/

Order by

- To change the default sort order from ascending to descending.

```
CREATE TABLE
crossfit_gyms_by_location
(  country_code text,
    state_province text,
    city text,
    gym_name text,
    PRIMARY KEY (country_code,
    state_province, city, gym_name)
) WITH CLUSTERING ORDER BY
  (state_province DESC, city ASC,
   gym_name ASC);
```

- `crossfit.cql` contains DDL and DML for this example.

Skinny row vs. Wide row

- With C*, you have a decision to make about the size of rows: skinny row vs. wide row
  - Skinny row: A fixed number of columns – closer to RDBMS
  - Wide row: lots of columns (e.g. tens of thousands or even millions)
- Primary Keys determine either skinny or wide rows:
  - If the primary key only contains the partition key then the table has skinny rows
  - If the primary key contains clustering columns then the table has wide rows

Skinny rows

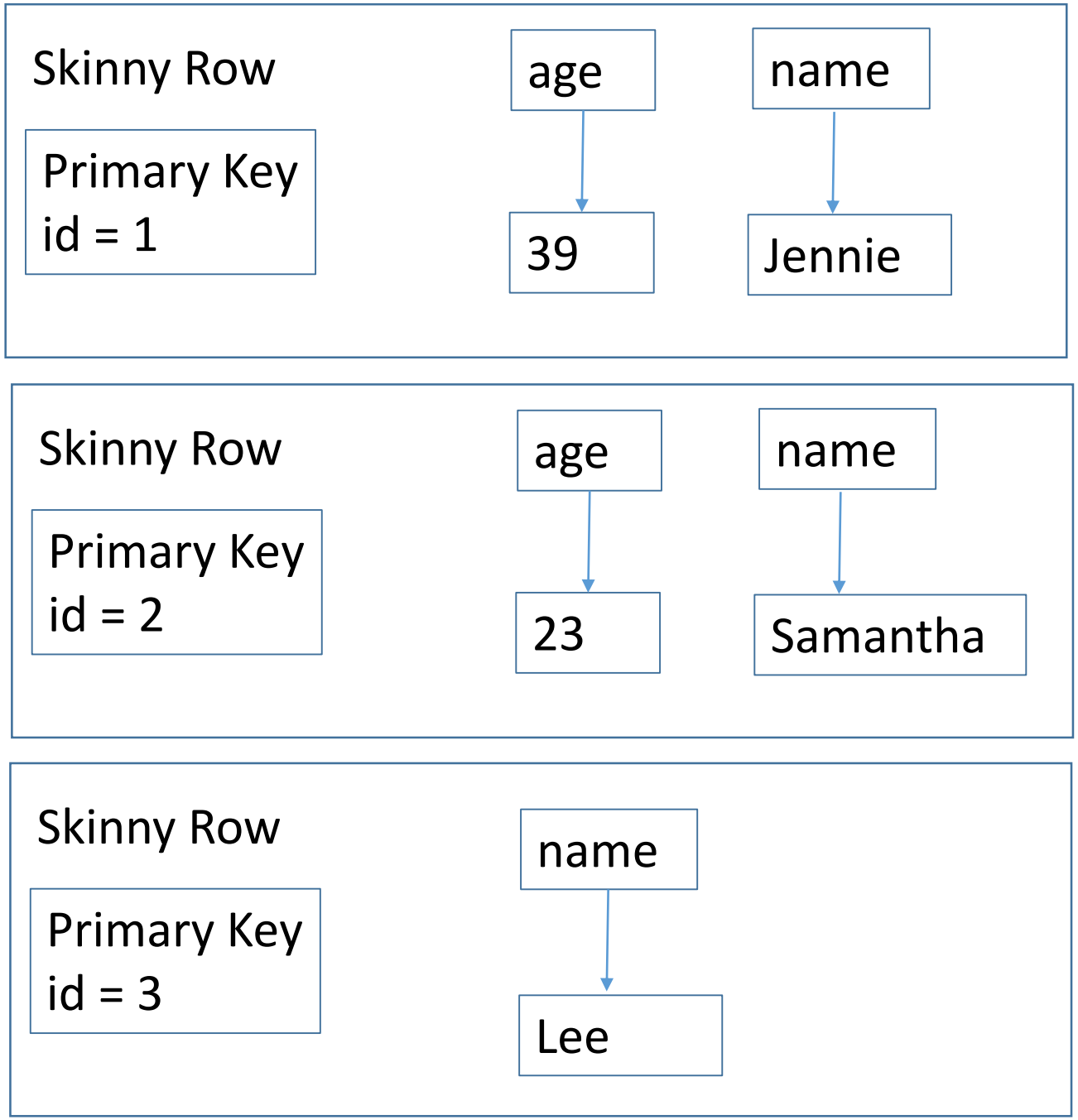
```
CREATE KEYSPACE IF NOT EXISTS
mykeyspace
WITH REPLICATION =
{'class':'SimpleStrategy',
'replication_factor':3};


use mykeyspace;
CREATE TABLE demo_skinny (id INT, name
VARCHAR, age INT, PRIMARY KEY(id));
INSERT INTO demo_skinny (id, name, age)
VALUES (1, 'Jennie', 39);
INSERT INTO demo_skinny (id, name, age)
VALUES (2, 'Samantha', 23);
INSERT INTO demo_skinny (id, name)
VALUES (3, 'Lee');
```

```
cqlsh:mykeyspace> select * from demo_skinny;

 id | age  | name
----+------+----------
  1 |   39 |   Jennie
  2 |   23 | Samantha
  3 | null |      Lee
```

skinny_rows.cql



demo_skinny table

Instead of storing null, which would waste space, the corresponding column will not be stored at all .

A table containing skinny rows

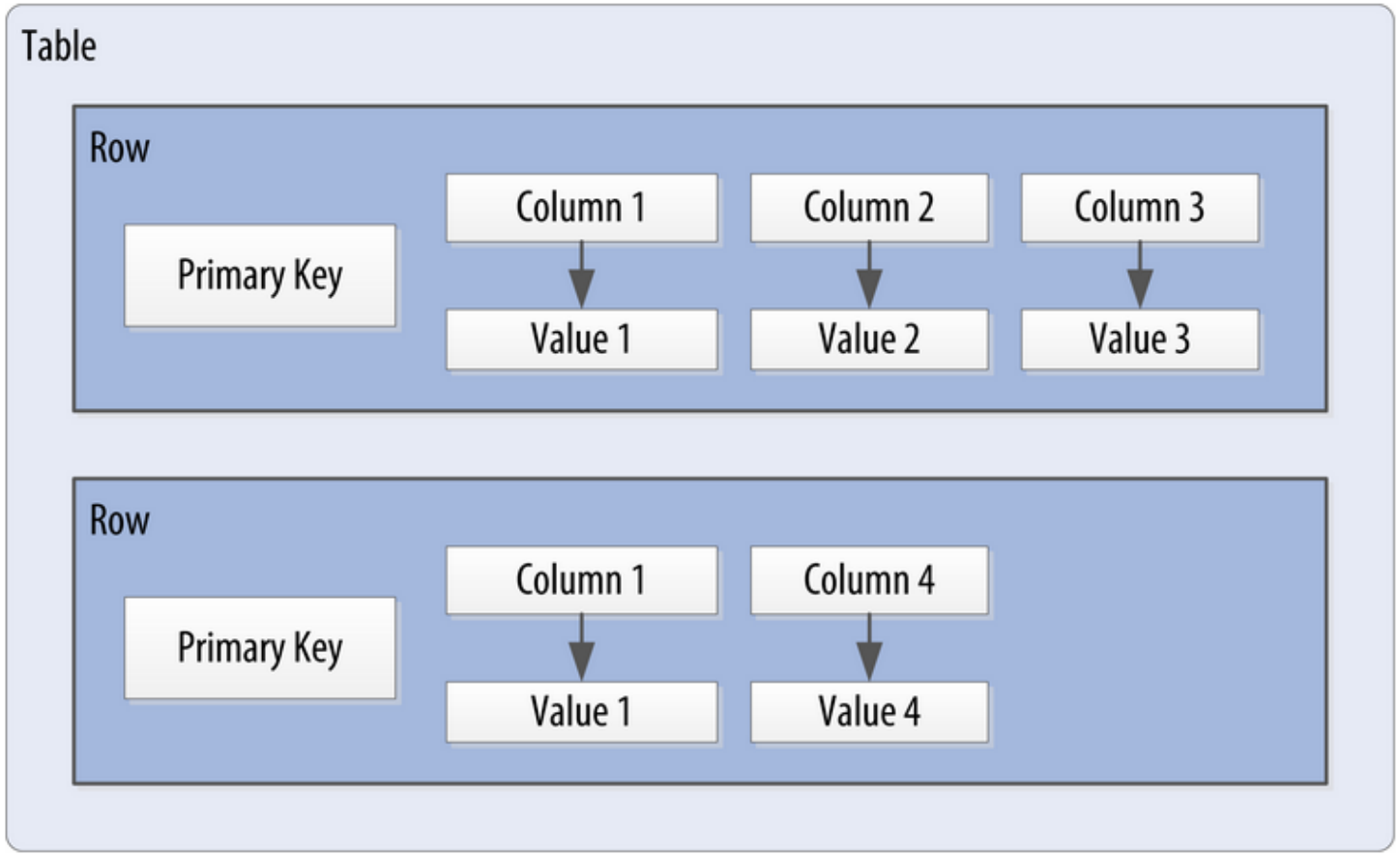


Figure 4-4. A Cassandra table

Wide rows

```
CREATE KEYSPACE IF NOT EXISTS mykeyspace
WITH REPLICATION = {'class':'SimpleStrategy', 'replication_factor':3};

use mykeyspace;
CREATE TABLE demo_wide
(id TIMESTAMP, city VARCHAR, hits COUNTER, PRIMARY KEY (id,city))
WITH COMPACT STORAGE;

UPDATE demo_wide SET hits = hits + 1 WHERE id = '2016-01-09+0000' AND city = 'NY';
UPDATE demo_wide SET hits = hits + 5 WHERE id = '2016-01-09+0000' AND city = 'Bethesda';
UPDATE demo_wide SET hits = hits + 2 WHERE id = '2016-01-09+0000' AND city = 'SF';
UPDATE demo_wide SET hits = hits + 3 WHERE id = '2016-01-10+0000' AND city = 'NY';
UPDATE demo_wide SET hits = hits + 1 WHERE id = '2016-01-10+0000' AND city = 'Baltimore';
```

wide_rows.cql

```
cqlsh:mykeyspace> select * from demo_wide;

 id                              | city      | hits
---------------------------------+-----------+------
 2016-01-10 00:00:00.000000+0000 | Baltimore |    1
 2016-01-10 00:00:00.000000+0000 |        NY |    3
 2016-01-09 00:00:00.000000+0000 |  Bethesda |    5
 2016-01-09 00:00:00.000000+0000 |        NY |    1
 2016-01-09 00:00:00.000000+0000 |        SF |    2

(5 rows)
cqlsh:mykeyspace> select * from demo_wide where id = '2016-01-10-0000';

 id                              | city      | hits
---------------------------------+-----------+------
 2016-01-10 00:00:00.000000+0000 | Baltimore |    1
 2016-01-10 00:00:00.000000+0000 |        NY |    3

(2 rows)
```

A table containing wide rows

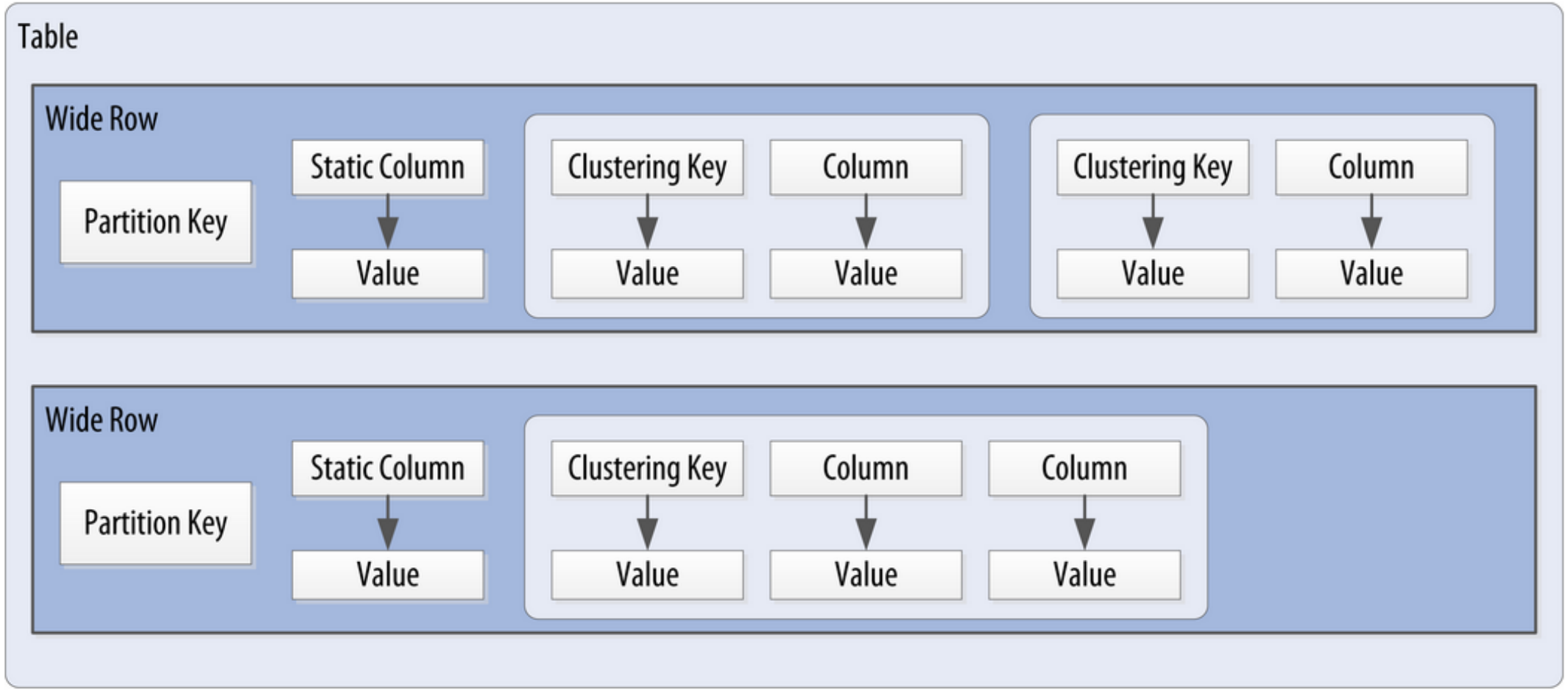
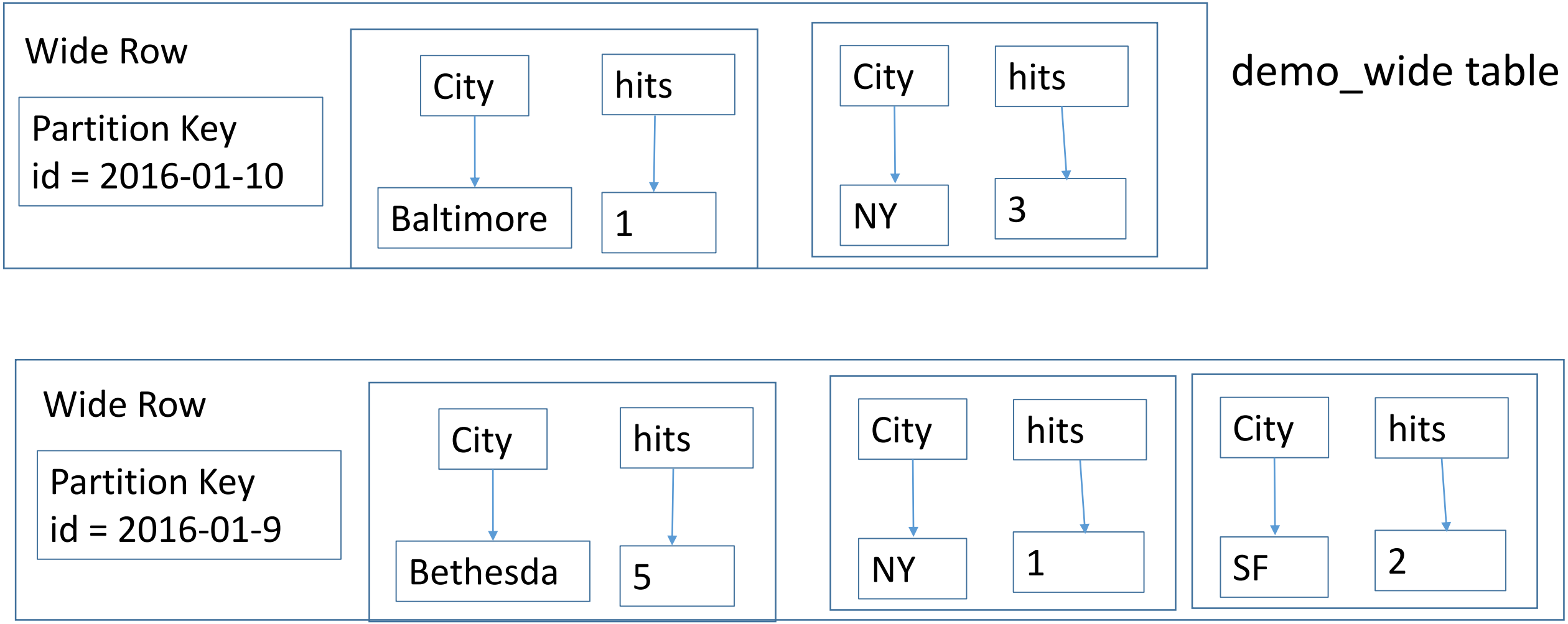


Figure 4-5. A Cassandra wide row

Static Column: (only) static within a given partition.

```
CREATE TABLE t (
  k text,
  s text STATIC,
  i int,
  PRIMARY KEY (k, i)
);
INSERT INTO t (k, s, i) VALUES ('k', 'shared', 0);
INSERT INTO t (k, s, i) VALUES ('k', ‘still shared', 1);
SELECT * FROM t;
```

```
k |                 s       | i
--------------------------------
k  | "still shared" | 0
k  | “still shared"   | 1
```


Wide Row
Partition Key
id = 2016-01-10
City
hits
Baltimore
1
City
hits
NY
3
demo_wide table
Wide Row
Partition Key
id = 2016-01-9
City
hits
Bethesda
5
City
hits
NY
1
City
hits
SF
2

Another example of wide rows

```
CREATE TABLE data (
  sensor_id int,
  collected_at timestamp,
  volts float,
  PRIMARY KEY (sensor_id, collected_at)
) WITH COMPACT STORAGE;
```

```
SELECT * FROM data;

 sensor_id | collected_at            | volts
-----------+-------------------------+-------
         1 | 2013-06-05 15:11:00-0500 |   3.1
         1 | 2013-06-05 15:11:10-0500 |   4.3
         1 | 2013-06-05 15:11:20-0500 |   5.7
         2 | 2013-06-05 15:11:00-0500 |   3.2
         3 | 2013-06-05 15:11:00-0500 |   3.3
         3 | 2013-06-05 15:11:10-0500 |   4.3
```

Special columns: Timestamps

- Generated for each column value that is updated.

```
cqlsh:mykeyspace> SELECT age, name, writetime(name) FROM
demo_skinny;
age  | name          | writetime(name)
------+-------------+------------------
   50 |   Jennie    | 1484006524137322
   23 | Samantha    | 1484006524216602
 null |       Lee   | 1484006547752743
```

- C* uses these timestamps for resolving any conflicting changes that are made to the same value. (The last timestamp wins in general.)
- The primary key columns will not have the timestamp

```
cqlsh:mykeyspace> SELECT id, writetime(id) FROM demo_skinny;
InvalidRequest: Error from server: code=2200 [Invalid query]
message="Cannot use selection function writeTime on PRIMARY KEY
part id".
```

Special columns: Time To Live (TTL)

- TTL is a value that C* stores for each column value to indicate how long to keep the value. By default, its value is null meaning the column will not expire.
- No TTL can be set to a primary key.
- The TTL can only be set for a column with a value.

```
cqlsh:mykeyspace> UPDATE demo_skinny USING TTL 60 SET age = 60 WHERE id = 3;
cqlsh:mykeyspace> SELECT id, age, name, TTL (age) from demo_skinny WHERE id = 3;

 id | age | name | ttl(age)
----+-----+------+----------
  3 |  60 |  Lee |       20

cqlsh:mykeyspace> SELECT id, age, name, TTL (age) from demo_skinny WHERE id = 3;

 id | age  | name | ttl(age)
----+------+------+----------
  3 | null |  Lee |     null
```

CQL Numeric Data Types

Types	Descriptions
int	A 32-bit signed integer (as in Java)
bigint	A 64-bit signed long integer (equivalent to a Java long)
smallint	A 16-bit signed integer (equivalent to a Java short)
tinyint	An 8-bit signed integer (as in Java)
varint	A variable precision signed integer (equivalent to java.math.BigInteger)
float	A 32-bit IEEE-754 floating point (as in Java)
double	A 64-bit IEEE-754 floating point (as in Java)

CQL Textual Data Types

Types	Descriptions
text, varchar	A UTF-8 character string
ascii	An ASCII character string

Time and Identity Data Types

	Descriptions
timestamp	timestamp as the value of a column itself
date, time	
uuid (universally unique identifier)	Type 4 UUID Dash-separated sequence of hex digits (e.g. **123e4567-e89b-12d3-a456-426655440000)**
timeuuid	Type 1 UUID (more frequently used than uuid due to the availability of convenience functions)

Other Simple Data Types

Types	Descriptions
boolean	true/false
blob	Binary large object – arbitrary array of bytes
inet	IPv4 or IPv6 Internet Address
counter	64-bit signed integer, its value cannot set directly, but only incremented or decremented.

Collections: set, list, and map

- Unordered collections. (cqlsh returns the elements in sorted order.)
- The ability to insert additional items without having to read the contents first.
- Set operations: `+, -, = { }`

```
cqlsh:mykeyspace> ALTER TABLE demo_skinny ADD emails set<text>;
cqlsh:mykeyspace> UPDATE demo_skinny SET emails = {'jennie@gmail.com'}
WHERE id = 1;
cqlsh:mykeyspace> UPDATE demo_skinny SET emails = emails + {'jsmith@sjsu.edu'}
WHERE id = 1;
cqlsh:mykeyspace> select * from demo_skinny where id = 1;
 id | age | emails                                  | name
----+-----+-----------------------------------------+--------
  1 |  50 | {'jennie@gmail.com', 'jsmith@sjsu.edu'} | Jennie
```

Collections: set, list and map

- Ordered list of elements – can reference an element by index
- Operations: +, -, [i]

```
 cqlsh:mykeyspace> ALTER TABLE demo_skinny ADD phone_numbers list<text>;
cqlsh:mykeyspace> UPDATE demo_skinny SET phone_numbers=phone_numbers+['1-408-678-9012'] where id = 2;
cqlsh:mykeyspace> select * from demo_skinny where id = 2;
 id | age  | emails | name     | phone_numbers
----+------+--------+----------+-----------------------------------
2 |   23 |   null | Samantha | ['1-408-890-1234', '1-408-678-9012']
cqlsh:mykeyspace> DELETE phone_numbers[0] from demo_skinny WHERE id = 2;
cqlsh:mykeyspace> select * from demo_skinny;
 id | age  | emails | name     | phone_numbers
----+------+--------+----------+------------------
  1 |   50 |   null |   Jennie |             null
  2 |   23 |   null | Samantha | ['1-408-678-9012']
  3 | null |   null |      Lee |             null
```

Collections: set, list and map

- A collection of key/value pairs
- The key and value can be any type except for counter.

cqlsh:mykeyspace> ALTER table demo_skinny ADD login_sessions map<timeuuid,int>;

cqlsh:mykeyspace> UPDATE demo_skinny SET login_sessions = {now():13, now():18} WHERE id = 1;

cqlsh:mykeyspace> select * from demo_skinny WHERE id = 1;

```
cqlsh:mykeyspace> select * from demo_skinny WHERE id = 1;

 id | age | login_sessions
                  | name
----+-----+-----------------------------------------------------------------
-----------------+--------
  1 |  39 | {9a72e190-1659-11e7-a845-f1c4fe027c28: 18, 9a72e191-1659-11e7-a845-f
1c4fe027c28: 13} | Jennie
```

Collections: set, list and map

cqlsh:mykeyspace> update demo_skinny set login_sessions[9a72e190-1659-11e7-a845-f1c4fe027c28] = 100 where id = 1;

```
cqlsh:mykeyspace> select * from demo_skinny WHERE id = 1;

 id | age | login_sessions
                    | name
----+-----+-----------------------------------------------------------------
------------------+--------
  1 |  39 | {9a72e190-1659-11e7-a845-f1c4fe027c28: 100, 9a72e191-1659-11e7-a845-
f1c4fe027c28: 13} | Jennie
```

User-Defined Types (UDT)

cqlsh:mykeyspace> CREATE TYPE address(street text, city text, state text, zip_code int);

cqlsh:mykeyspace> ALTER TABLE demo_skinny ADD addresses map<text, address>;

InvalidRequest: Error from server: code=2200 [Invalid query] message="Non-frozen UDTs are not allowed inside collections: map<text, address>"

cqlsh:mykeyspace> ALTER TABLE demo_skinny ADD addresses map<text, frozen<address>>;

Note: C* considers UDT as a collection. You can nest a collection within another collection by marking it as frozen.

User-Defined Types (UDT)

```
cqlsh:mykeyspace> UPDATE demo_skinny SET addresses = addresses +
{'home':{street: '7712 Broadway', city:'Tucson', state:'AZ', zip_code:12345}}
WHERE id = 1;

cqlsh:mykeyspace> select * from demo_skinny where id = 1;


 id | addresses                                                                                                | age | name
----+----------------------------------------------------------------------------------------------------------+-----+--------
  1 | {'home': {street: '7712 Broadway', city: 'Tucson', state: 'AZ', zip_code:
12345}} |  50 | Jennie


(1 rows)
```

Secondary Index

- An index on a column that is not part of the primary key.

cqlsh:mykeyspace> select * from demo_skinny where name = 'Lee';

Will be an error without a secondary index on name.

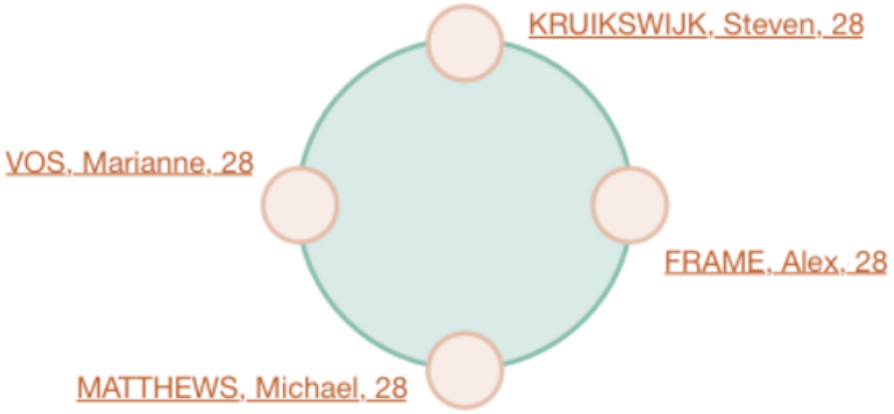
cqlsh:mykeyspace> CREATE INDEX ON demo_skinny (name);

cqlsh:mykeyspace> select * from demo_skinny where name = 'Lee';

```
 id | age  | name
----+------+------
  3 | null |  Lee
```

- It is also possible to create indexes that are based on values in collections.
- cqlsh:mykeyspace> DROP INDEX demo_skinny_name_idx;

Secondary Index



Example: A table storing cyclist names and ages using the last name of the cyclist as the primary key.

- Queries based on a particular range of last names, such as all cyclists with the last name Matthews will retrieve sequential rows from the table.
- A query based on the age, such as all cyclists who are 28, will require all nodes to be queried for a value, generally results in a prohibitive read latency and is not allowed.

Secondary Indexes

- Secondary indexes are used to query a table using a column that is not normally queryable due to a prohibitive read latency.
- Secondary indexes are stored locally on each node in a hidden table and built in a background process.
- A better solution is to create a materialized view.

Secondary Indexes are not recommended for ….

- Columns with high cardinality
  - Example: The majority of addresses are unique.
  - A query will incur many seeks for very few results.
- Columns with very low cardinality. e.g. title (such as Mrs., Mr., etc.)
- Columns that are frequently updated or deleted.

CQL has a number of rules for query predicates that ensure efficiency and scalability:

1. Only primary key columns may be used in a query predicate.
2. All partition key columns must be restricted by values (i.e. equality search).
3. All, some, or none of the clustering key columns can be used in a query predicate.
4. If a clustering key column is used in a query predicate, then all clustering key columns that precede this clustering column in the primary key definition must also be used in the predicate.
5. If a clustering key column is restricted by range (i.e. inequality search) in a query predicate, then all clustering key columns that precede this clustering column in the primary key definition must be restricted by values and no other clustering column can be used in the predicate.

Example schema

```
CREATE TABLE CROSSFIT (
    country_code text,
    state_province text,
    city text,
    gym_name text,
    since int,
    PRIMARY KEY (country_code, state_province, city, gym_name,since)
) WITH CLUSTERING ORDER BY (state_province DESC, city ASC,
gym_name ASC);
```

2. `select * from crossfit where state_province = 'NY';`

InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING

4. `select * from crossfit where city = 'Toronto';`

InvalidRequest: Error from server: code=2200 [Invalid query] message="PRIMARY KEY column "city" cannot be restricted as preceding column "state_province" is not restricted“

5-1. `select * from crossfit where country_code = 'USA' and state_province = 'NY' and city > 'A' and city < 'C';`

```
country_code | state_province | city     | gym_name          | since
-------------+----------------+----------+-------------------+-------
         USA |             NY | Brooklyn | CrossFit Brooklyn |  2012
```

Error if country_code and state_province values are not specified.

5-2 select * from crossfit where country_code = 'USA' and state_province = 'NY' and city > 'A' and city < 'C' and since > 2015;

InvalidRequest: Error from server: code=2200 [Invalid query] message="Clustering column "since" cannot be restricted (preceding column "city" is restricted by a non-EQ relation)"