Cassandra Fundamentals

Module 3

CQL 3: Cassandra Query Language

Module plan

- CQL review
- Data types
- Collections
- Nesting
- Queries
- Transactions

CQL

- Query Language for C*
- Closely resembles SQL
- Main Query Interface
  - old alternative: Thrift RPC (deprecated)
- Performant
- Rather mature: since C* 0.8.0 (3+ years)
- CQL current version: 3.2.? (no backward compatibility to CQL 2)

CQL Versions

- Pre-CQL (Thrift): 2008
- CQL 1: Cassandra 0.8: 2011
- CQL 2: Cassandra 1.0: 2012
  - No support for Composite types
- CQL 3.1: Cassandra 1.1 (beta), 1.2 (default), 2.0 : 2013
  - Collections, Atomic batches, system tables
  - Denormalized tables
- CQL 3.2: Cassandra 2.1: 2014

Keywords and Identifiers

- Keywords are case-isensitive
- Identifiers are autoconverted to lower case
  - exception: “double QuOtEd identifiers”

CQL Shell

- cqlsh – python-based command-line client
  - prereq: python 2.7.5+
  - prereq: python thrift module
  - install python CQL module
    - pylib/python setup.py install
- unix: bin/cqlsh <host> <port>
- windows: python cqlsh <host> <port>
- default: localhost 9160

CQLSH Basic Keyspaces Management

- CREATE KEYSPACE mykeyspace …
- USE mykeyspace;
- DESCRIBE KEYSPACE mykeyspace;
  - prints command that created mykeyspace
- SELECT * FROM system.schema_keyspaces;
  - definitions of all keyspaces
- DROP KEYSPACE mykeyspace;
- ALTER KEYSPACE mykeyspace …

NODETOOL REPAIR required

CQLSH Basic Table Management

- CREATE TABLE t1 (id varchar, s1 int, s2 int, PRIMARY KEY(id));
- INSERT INTO t1(id,s1,s2) VALUES (‘id1’,10,20);
- DESCRIBE TABLE t1;
- SELECT * FROM t1;
- SELECT * FROM t1 WHERE id=‘id1’;

PRIMARY KEY!

- DELETE FROM t1 WHERE id=‘id2’;
- DROP TABLE t1;
- TRUNCATE t1;

CQLSH Basic Indexing

- SELECT * FROM t1 WHERE s2=20;

  NOT PRIMARY KEY

  - Bad Request: No indexed columns present…
- CREATE INDEX ON t1(s2);
  - Hidden synchronized table t1_s2_idx
  - Can be discovered using DESCRIBE TABLE <base_table>
- Multiple indexes are allowed
- SELECT * FROM t1 WHERE s1=10 and s2=20 ALLOW FILTERING;

  Agree with possible performance degradation

Cassandra Data Types

- String
- Numeric
- UUIDs
- Collections
- Miscelaneous

- Used for input validation, ordering, client library interaction

Cassandra String Data Types

- ASCII
  - Single-byte US-ASCII coded string (bytes 0-127)
- TEXT (=VARCHAR)
  - UTF-8 encoded string
- INET
  - IPv4 or IPv6 address in a string format

Cassandra Numeric Data Types

INTEGERS

- INT 32-bit signed
- BIGINT 64-bit signed
- VARINT Arbitrary precision integer
  - java.math.BigInteger

SPECIAL INTEGERS

- COUNTER 64-bit integer distributed counter

FLOATING POINT NUMBERS

- DECIMAL Variable precision fixed-point numeric
  - java.math.BigDecimal
- FLOAT 32-bit IEEE-754 floating point number
  - java.lang.Float
- DOUBLE 64-bit IEEE-754 floating point number
  - java.lang.Double

Cassandra UUID Data Types

- UUID
  - Standard UUID Type 1: high-precision timestamp and MAC address
  - Standard UUID Type 4: (pseudo) random numbers
  - literal: hexadecimal digits 8-4-4-4-12
- TIMEUUID
  - Type 1 UUID for storing unique time-based IDs
  - Can be converted between UUIDs and timestamps

Cassandra Data Type for Dates and Times

- TIMESTAMP
  - Date+time only (no separate type for dates and times)
  - 8 bytes
  - Milliseconds since midnight UTC on 01.01.1970
  - literal: ‘yyyy-mm-dd HH:mm:ssZ’
  - literal: number of milliseconds since epoch

Cassandra Miscellaneous Data Types

- BOOLEAN
  - Literal: true (no quotation marks)
  - Literal: false (no quotation marks)
- BLOB
  - Binary data in hex
  - Prefixed with 0x
  - No quotation marks
- TUPLE
  - A group of fields (up to 32768)
  - C* 2.1+

Cassandra Collection Types

- LIST
  - values in insertion order; duplicates are allowed
  - [‘Moscow’,’SanktPeterburg’]
- SET
  - unordered unique values; returns in natural ordering
  - {‘cql’,’thrift’,’avro’}
- MAP
  - key-value pairs with unique keys
  - {‘1’:’cql1’, ‘2’:’cql2’, ‘3’:’cql3’}

Using Collection Types

- ALTER TABLE t1 ADD s3 list<text>;
- UPDATE t1 SET s3=[‘a’,’b’,’c’] WHERE id=‘id10’;
- CREATE INDEX ON t1(s3); C* 2.1+

Cassandra Collections - Set Operations

- Adding an element to the set

```
UPDATE collections_example
SET set_example = set_example + {'3-three'} WHERE id = 1;
```

- After adding this element, it will sort to the beginning.

```
UPDATE collections_example
SET set_example = set_example + {'0-zero'} WHERE id = 1;
```

- Removing an element from the set

```
UPDATE collections_example
SET set_example = set_example - {'3-three'} WHERE id = 1;
```

Cassandra Collections - List

- Ordered by insertion

```
list_example list<text>
```


```
INSERT INTO collections_example (id, list_example)
VALUES(1, ['1-one', '2-two']);
```

Cassandra Collections - List Operations

- Adding an element to the end of a list

```
UPDATE collections_example
SET list_example = list_example + ['3-three'] WHERE id = 1;
```

- Adding an element to the beginning of a list

```
UPDATE collections_example
SET list_example = ['0-zero'] + list_example WHERE id = 1;
```

- Deleting an element from a list

```
UPDATE collections_example
SET list_example = list_example - ['3-three'] WHERE id = 1;
```

Cassandra Collections - Map Operations

- Add an element to the map

```
UPDATE collections_example
SET map_example[3] = 'three' WHERE id = 1;
```

- Update an existing element in the map

```
UPDATE collections_example
SET map_example[3] = 'tres' WHERE id = 1;
```

- Delete an element in the map

```
DELETE map_example[3]
FROM collections_example WHERE id = 1;
```

Counter Type

- Only increment/decrement
- Should NOT be a primary key
- Should be the only column in a table besides primary key
- Not indexable
- Cannot be set to expire using TTL

UUID Types

- 128-bit number
- 32 hex-digits separated by dashes after 8,12,16,20
- TIMEUUID
  - Variant 1 UUID: MAC address and date/time
  - 60 bit: time in 100 nanosecond resolution since 00:00:00:00.00 UTC
  - 14 bit: clock sequence number
  - 48 bit: MAC address
  - sorted by timestamp

UUID functions

- dateOf()
  - extracts timestamp from TIMEUUID as a date
- unixTimestampOf()
  - extracts timestamp from TIMEUUID as milliseconds
- now()
  - generates new unique TIMEUUID in UTC
- minTimeuuid() and maxTimeuuid()
  - for inequality checks
  - returns an UUID-like result given a time
  - SELECT * FROM t1 WHERE t>maxTimeuuid('2014-12-12 00:05+0000) AND t<minTimeuuid('2014-12-20 00:00+0000');

Timestamp Type

- 64-bit number of milliseconds since epoch (01.01.1970 00:00:00 GMT)
- Formats (Z – RFC-822 4-digit time zone: +0001, -0003)
  - yyyy-mm-dd[Z]
  - yyyy-mm-dd HH:mm[:ss][Z]
  - yyyy-mm-dd’T’HH:mm[:ss][Z]

Tuple Type

- An alternative for user-defined types at prototyping
- CREATE TABLE t2 (id int PRIMARY KEY,
  v frozen <tuple<int, text, float>>);
- INSERT INTO t2(id,v) VALUES (0, (3,’abc’,4.3));
- Can be indexed
- Can be nested
  - CREATE TABLE t3 (id int PRIMARY KEY,
    v frozen <tuple<int, tuple<text, float>>>);

User-defined Type

- CREATE TYPE IF NOT EXISTS mykeyspace.newtype (field1 int, field2 text, field3 double);
- CREATE TABLE t4 (id text PRIMARY KEY, v frozen <newtype>, v2 map<text, frozen <newtype>>);
- ALTER TYPE newtype ADD field4 timeuuid;
- ALTER TYPE newtype RENAME field3 TO f3;
- ALTER TYPE newtype ALTER field2 TYPE blob;
- DROP TYPE newtype;
- DESCRIBE TYPE newtype;

CREATE KEYSPACE (CQL 3 for C* 1.2+)

```
CREATE KEYSPACE “MySpace”
with REPLICATION = {‘class’: ‘SimpleStrategy’, ‘replication_factor’: 2}
and DURABLE_WRITES = TRUE;
```

```
CREATE KEYSPACE “MySpace”
with REPLICATION = {‘class’: ‘NetworkTopologyStrategy’, ‘dc2’: 1,
‘dc1’: 2};
```

CREATE KEYSPACE (CQL 3 for C* up to 1.1)

```
CREATE KEYSPACE “MySpace”
with STRATEGY_CLASS = SimpleStrategy
and STRATEGY_OPTIONS:REPLICATION_FACTOR = 2
and DURABLE_WRITES = TRUE;
```

```
CREATE KEYSPACE “MySpace”
with STRATEGY_CLASS = NetworkTopologyStrategy
and STRATEGY_OPTIONS:DC1 = 2 and STRATEGY_OPTIONS:DC2 = 1;
```

Replication Strategies

- SimpleStrategy
  - First replica – according to partitioner
  - Next replicas – on next nodes (on one replica on one node) clockwise in the ring

For single data center cluster

- NetworkTopologyStrategy
  - Replication factors are individual for different DCs

    {class: ‘NetworkTopologyStrategy’, ‘us-east’:2; ‘us-west’: 3}
  - Each data center have the complete data
  - First replica for each data center – according to partitioner
  - Next replicas in each data center – on next nodes clockwise (if possible, at different racks)

For multiple data centers or racks

Replication Factor Considerations

- Reads should be served from local data center
- Replication Factor should be <= number of nodes in the cluster
  - otherwise writes are rejected
    - reads can still be served
- Minimize affection of failures
- Replication Factor update is possible
  - leads to re-distribution of replicas

Creating Tables

```
CREATE TABLE “Users” (
        “username” text PRIMARY KEY,
        “email” text,
        “pass” blob
);
```

- No NULLs (and “NOT NULL” constraints)
- No default values
- No data validation

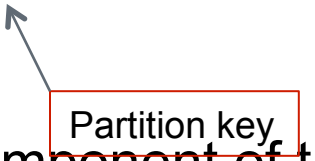
PRIMARY KEY options

```
CREATE TABLE “Users” (
        “username” text PRIMARY KEY,
        “email” text,
        “pass” blob
);

CREATE TABLE “Users” (
        “username” text,
        “email” text,
        “pass” blob,
        PRIMARY KEY(username)
);
```

Primary Key in Partitioning

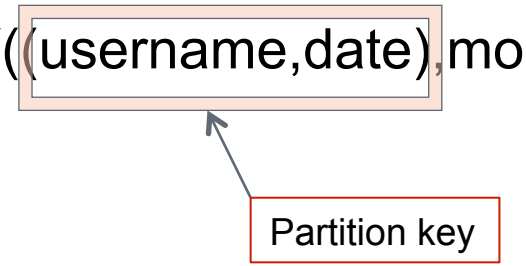
```
CREATE TABLE “UsersActions” (
        “username” text,
        “moment” timestamp,
        “details” blob,
        PRIMARY KEY (username, moment)
);
```



- Partition key is the first component of the primary key
- Murmur3(username) => active node for that row

Composite Partition Key

```
CREATE TABLE “UsersActions” (
        “username” text,§
        “date” text,
        “moment” timestamp,
        “details” blob,
        PRIMARY KEY((username,date),moment)
);
```



Methods for writing data

- INSERT INTO
  - literals
- COPY
  - from .csv file
- sstableloader

Data Insertion

```
INSERT INTO “Users” (“username”,“email”,“pass”)
VALUES (‘abreiman’,’abreiman@luxoft.com’,0x94a6b2d90002bcaaff);
```

```
COPY “Users” (“username”,”email”,”pass”)
FROM ‘/path/to/file.csv’
WITH header=true AND delimiter=‘;’;
```

- Successful write is silent
  - exception: lightweight transactions (conditional INSERT or UPDATE)
- Write failure results in error message

Upsert: inserting existing row

```
INSERT INTO “Users” (“username”,“email”,“pass”)
VALUES (‘abreiman’,’abreiman@luxoft.com’,0x94a6b2d90002bcaaff);

INSERT INTO “Users” (“username”,“email”)
VALUES (‘abreiman’,’abreyman@luxoft.com’);
```

Effectively UPDATE

- No error (primary key violation)
- Silently updates existing row

Lightweight transaction – conditional insert

- To change default upsert-like behavior into relational-like:

```
INSERT INTO “Users” (“username”,“email”)
VALUES (‘abreiman’,’abreyman@luxoft.com’)
IF NOT EXISTS;
```

IF => PAXOS
Latency cost!

- If not exists, returns positive result row (exception from silent success rule)

```
[applied]
True
```

- If exists, returns existing row

```
[applied] | username | email                | pass
False     | abreiman | abreiman@luxoft.com | 0x94a6b2d90002bcaaff
```

Updating Data

```
UPDATE “Users”
SET pass=0x03982482394
WHERE username=‘abreiman’;
```

Single column update

```
UPDATE “Users”
SET pass=0x03982482394,
      email=‘abreiman2@luxoft.com’
WHERE username=‘abreiman’;
```

Multiple columns update

Upsert: updating not existing row

```
UPDATE “Users”
SET pass=0x03982482394
WHERE username=‘abreiman’;
```

Effectively INSERT

```
username | email | pass
abreiman  | null    | 0x03982482394
```

Lightweight transaction – conditional update

ALTER TABLE “Users” ADD “version” timeuuid;

UPDATE “Users”
SET version = 12345678-1234-11a4-8eeb-12345678abcd; -- NOW();

UPDATE “Users”
SET pass=0x03982482394, version=NOW()
WHERE username=‘abreiman’
IF version = 12345678-1234-11a4-8eeb-12345678abcd;

Optimistic locking with PAXOS

- If version is the same, returns positive result row (exception from silent success rule)

[applied]

True

- If version is not the same, returns values from row

[applied]| username | email | pass | version

False | abreiman | abreiman@luxoft.com | 0x94a6 | 87654321-4321-11a4-8eeb-12345678abcd

Updating Data with Time-to-live

```
UPDATE “Users”
USING TTL 3600
SET pass=0x03982482394
WHERE username=‘abreiman’;
```

- TTL is set in seconds
- After expiry set column value(s) (or whole row) will be deleted (tombstoned)

Deleting Data

- DELETE
  - to delete a value in a column, row or rows
- TRUNCATE
  - delete all rows
- DROP
  - delete a table or a keyspace

Deleting Data

```
DELETE pass
FROM “Users”
WHERE username=‘abreiman’;
```

```
DELETE
FROM “Users”
WHERE username=‘abreiman’;
```

```
TRUNCATE “Users”;
```

```
DROP TABLE “Users”;
```

Deleting Data Process

- DELETE creates a tombstone
- Actual delete (compaction) – not before **gc_grace_seconds** will pass
  - default: 10 days (864000)
- Compaction combines SSTables
  - automatically
  - using ‘nodetool compact’

Batching

- BATCH combines multiple writes (INSERT, UPDATE, DELETE) into one atomic logical unit
  - atomicity only
  - no isolation

```
BEGIN [UNLOGGED] BATCH
[USING TIMESTAMP ts1
INSERT … [USING TIMESTAMP ts2];
UPDATE … [USING TIMESTAMP ts3];
APPLY BATCH;
```

Good Practice In Batching

```
BEGIN UNLOGGED BATCH;

INSERT INTO weather_readings (date, timestamp, temp) values
(20140822,'2014-08-22T11:00:00.00+0000', 98.2);

INSERT INTO weather_readings (date, timestamp, temp) values
(20140822,'2014-08-22T11:00:15.00+0000', 99.2);

APPLY BATCH;
```

- Same partition key => one write

Another Good Practice In Batching

```
BEGIN BATCH;
UPDATE users SET middlename=’D.’ where username = ‘abreiman’;
UPDATE users_by_inn SET fullname=‘Alexander D. Breiman’ where
inn=‘1234567890’;
APPLY BATCH;
```

- Keeping two tables in sync

Bad Practice In Batching

```
BEGIN BATCH;
INSERT INTO t1 VALUES (1);
INSERT INTO t1 VALUES (2);
INSERT INTO t1 VALUES (3);
APPLY BATCH;
```

- Work more for nothing
- No need in batching

Updating unordered unique collections : Sets

ALTER TABLE “Users” ADD “starred_by” **SET<text>**;

UPDATE “Users”
SET “starred_by” = “starred_by” + **{**‘vsonkin’,’ptsytovich’**}**
WHERE username=‘abreiman’;

UPDATE “Users”
SET “starred_by” = “starred_by” + **{**‘vsonkin’**}**
WHERE username=‘abreiman’;

No dups: set

UPDATE “Users”
SET “starred_by” = “starred_by” – **{**’ptsytovich’**}**
WHERE username=‘abreiman’;

- Safe (atomic)
- Upsert semantics

Updating ordered non-unique collections: Lists

ALTER TABLE “Users” ADD “mentioned_by” **LIST<text>**;

```
UPDATE “Users”
SET “mentioned_by” = “mentioned_by” + [‘vsonkin’]
WHERE username=‘abreiman’;
```

Add to tail

```
UPDATE “Users”
SET “mentioned_by” = [‘vsonkin’] + “mentioned_by”
WHERE username=‘abreiman’;
```

Add to head

```
UPDATE “Users”
SET “mentioned_by”[0] = ‘ptsytovich’
WHERE username=‘abreiman’;
```

Change by (zero-based) index
Error if out of list bounds!

Deleting From Lists

```
UPDATE “Users”
SET “mentioned_by” = “mentioned_by”- [‘vsonkin’]
WHERE username=‘abreiman’;
```

```
DELETE “mentioned_by”[0]
FROM “Users”
WHERE username=‘abreiman’;
```

Updating Key-Value Pairs Collections: Maps

ALTER TABLE “Users” ADD “identities” **MAP<text,bigint>**;

UPDATE “Users”
SET “identities” = **{‘twitter’:12345, ‘instagram’:54321}**
WHERE username=‘abreiman’;

UPDATE “Users”
SET “identities”[‘instagram’] = 98765, “identities”[‘facebook’]=111111
WHERE username=‘abreiman’;

DELETE “identities”[‘instagram’]
FROM “Users”
WHERE username=‘abreiman’;

Secondary Indexes and Selection From Collections

```
CREATE INDEX ON “Users” (“starred_by”);  -- Set
CREATE INDEX ON “Users” (KEYS(“identities”));  -- Map
```

```
SELECT *
FROM “Users”
WHERE “starred_by” CONTAINS ‘vsonkin’;
```

```
SELECT *
FROM “Users”
WHERE “identities” CONTAINS KEY ‘twitter’;
```

Collections Limitations

- Collection is read as a whole; not possible to read element by index.
- Maximum 64Kb per collection (no error; auto truncation)
- Only one row to be read if WHERE key IN (x,y)
- Many rows can be read if no “IN”

Tuple inserting

ALTER TABLE “Users” ADD “education” **frozen<tuple<text, int>>**;

- **frozen** means indivisible: write all components together only
- C* 2.1: tuples must be declared frozen (nonfrozen planned for C* 3)
- Can be used like primitive type: in collections, nested tuples, PKs
- Can be (secondary) indexed: effectively multicolumn index

```
UPDATE “Users”
SET “education” = (‘University’, 1994)
WHERE “username”=‘abreiman’;
```

```
UPDATE “Users”
SET “education” = (‘University’, null)
WHERE “username”=‘abreiman’;
```

User-defined Types

- UDT is tuple with named components
- First-class entity within a keyspace (tuple defined for a single column)
- Indexable like a tuple

```
CREATE TYPE “education_data” (
  “school_name” text,
  “grad_year” int
);
```

```
ALTER TABLE “Users” ADD “education” frozen<“education_data”>;
```

```
UPDATE “Users”
SET “education” = {“school_name”: ‘University’, “grad_year”: 1994}
WHERE “username”=‘abreiman’;
```

```
SELECT “education”.”grad_year”
FROM “Users”
WHERE “username”=‘abreiman’;
```

Counters

- 64-bit signed integer value
- Can be only incremented or decremented (no direct set)
- Upsert semantics (0-based)
- No other datatypes (besides primary key) in counter table

Access Control: Users

CREATE USER adb WITH PASSWORD ‘123$123’ NOSUPERUSER;

LIST USERS; -- cqlsh

SELECT * FROM system_auth.users; -- username + super

SELECT * FROM system_auth.credentials; -- bcrypt(pass)

cqlsh -u adb -p 123$123

default user: cassandra:cassandra

- cassandra.yaml
  - authorizer: PasswordAuthorizer
  - authenticator: CassandraAuthenticator

Access Control: Grant

```
GRANT SELECT PERMISSION
ON KEYSPACE mykeyspace
TO adb;
```

```
GRANT MODIFY PERMISSION
ON TABLE mykeyspace.Users
TO adb;
```

```
SELECT * FROM system_auth.permissions; -- username + resource + perm
```

- Permissions
  - SELECT, MODIFY, CREATE, ALTER, DROP, AUTHORIZE
  - ALL

Access Control: Revoke

```
REVOKE MODIFY PERMISSION
ON KEYSPACE mykeyspace
FROM adb;
```

Thank you!

Questions?