- The official website of DataStax - http://www.datastax.com

- Cassandra High Availability - Robbie Strickland
Background

- Database Orientation
- What is Apache Cassandra
- Terminology
- Write Data Path
### Database Orientation

**Row-oriented:**

- Easy to act on full rows

**Column-oriented:**
- Without Remorse: 001, Patriot Games: 002
- Tom Clancy: {001, 002}

- Aggregate actions are easier
What is Apache Cassandra?

- A top-level open-source Apache project born at Facebook and built on Amazon’s Dynamo and Google’s BigTable.
- Distributed database for managing large amounts of structured data across many commodity servers, while providing highly available service and no single-point-of-failure.
Terminology used in this lecture

‘cassandra.yaml’ - Configuration file that describes the entire configuration of the node\cluster.

Node - Where you store your data. It is the basic infrastructure component of Cassandra.

Cluster - Collection of related nodes. the nodes share the cluster_name property in ‘cassandra.yaml’ and can come from different racks.

Rack - Unit of electronic equipment that is housed in the same framework. A few nodes can share a rack to accelerate their communication rate.

Data Center - Logical grouping of nodes inside the same cluster that share common characteristics (e.g., geographical location).
Write Data Path

- Commitlog - data is first written to the commitlog to ensure durability.
- Memtable - follows, it is written to an in-memory data structure called memtable.
- SSTable - once the memtable becomes full, the data is flushed to SSTable - Sorted String Table on disk (random I/O is avoided, as we write only baks).
- Writes are atomic at the row level - all columns are either fully written or not written at all.
- Coordinator - the node that receives the request. not necessarily the one that contains the data.
High Availability

- ACID
- CAP theorem
- Availability in Cassandra
- Common Architectures
- Cassandra’s Architecture
- **Atomicity** - Guarantees that database updates associated with a transaction occur in an *all-or-nothing* manner. If some part of the transaction fails, the state of the database remains unchanged.

- **Consistency** - Assures that any transaction will transfer the state of the system from one valid state to another (e.g., guarantee constraints enforcements).

- **Isolation** - Attempts to ensure that concurrent transactions that manipulate the same data do so in a controlled manner, essentially isolating in-process changes from other clients.

- **Durability** - Ensures that all writes are preserved in nonvolatile storage, most commonly on disk.
It is impossible for distributed computer system to simultaneously provide all three of these promises:

- **Consistency**: Assures that the integrity of data will be preserved across all instances of that data. Changes to a value in one location will definitely be reflected in all other locations (different from the ACID definition).
- **Availability**: When a node is in non-fault state, it will respond to a client’s request.
- **Partition-tolerant**: The distribution system continues to work despite a detachment between several nodes to others.

Cassandra tries to overcome this inability to satisfy all promises by leveraging a variety of consistency levels.
Availability in Cassandra

- In Cassandra, all aspects of ACID are continuum. In particular, the user can attach to a statement the level of consistency he desires.

- Cassandra presents a huge set of consistency levels for both read and write operations - we will further elaborate this subject.
Common Architectures - Monolithic Architecture

- One server that serves all applications.
- No coordination is required in order to fulfill ACID.
- Single point of failure.
- Multiple servers that serve all applications.
- The NAS is more robust and scalable (e.g., more storage).
- Single point of failure (just moved to another place).
Common Architectures - Master-Slave Architecture

- One node serves as master and only it can write. Other nodes are read-only replicas.
- Tries to address the consistency problem of distributed system.
- Single point of failure - at master node.
● Data partitioned into groups of keys (e.g., A-M in one cluster and N-Z in another).
● Knowledge of where key-request go rest in the application layer.
● Adding a shard requires manual shifting of data.
Some systems offer a layer that intermediate between application and server to avoid the need for application to know the key topology.

Multiple single point of failure - one for each master node.
When approaching the single-point-of-failure, the common solution is to employ a master failover protocol. Usually that means to appoint a new master.

This demands several things:
- The application needs to fully or partially understand the system’s topology.
- Data partition must be carefully planned.
- Availability is difficult to gain.
Cassandra’s Architecture

- Continuous consistency level.
- Availability as a key design consideration.
- All nodes are peers - all of them can either read and write.
- Nodes communicate via a ‘gossip’ protocol.
- Distributed Hash Tables to be used when node desires to find other node.
- Seed node - serves as a bootstrap coordinator to discover the cluster’s topology.
- Replicas - scattered all around the cluster to overcome the single-point-of-failure issue.
Data Distribution

- Hash Tables
- Distributed Hash Tables & Consistent Hashing
- Token Assignment & Vnodes
Hash Tables

- Regular hash table values are constructed by a fixed number of bits.
- Works well the number of slots is stable.
- Needed when a specific document is needed.
- Going over all nodes and ask them if they hold the document is:
  - Not scalable.
  - Demands the node to be fully available to answer such questions.
Distributed Hash Tables & Consistent hashing

- Applying the hash-table principles on a multi-node cluster -> Distributed Hash Table (DHT).
- Making a record’s hash value to be consistent and change-tolerant -> Consistent Hashing.
- Attach to each node hash-value that falls inside a ring.
- Every record’s hash-value falls in the same ring.
- A record is located in the closest node to it, clockwise.

Why every node has more than one slot? (8 in this example)
In this example, the node that contains the record "artist", "REM" is node 0.
Every node is aware of all others nodes’ ranges.

If a node falls down, it is easy to reassign all records that resides in it - only take care of documents before it on the ring.
● Until version 1.1: every node is responsible for one token = one range.
● From 1.2 - every node is responsible for several tokens = several ranges.
Token assignment & Vnodes

- Initial state
Token assignment & Vnodes

- Setting $num_{tokens} = 4$. 
Token assignment & Vnodes

- After shuffling.
Vnodes

Ring without virtual nodes

Node 1
A
F
E

Node 2
B
A
F

Node 3
C
B
A

Node 4
D
C
B

Node 5
E
D
C

Node 6
F
E
D

Ring with virtual nodes

Node 1
B
E
G
K
D
J
L
A

Node 2
A
P
M
O
D
H
K
F

Node 3
K
G
C
N
J
F
P
I

Node 4
M
O
I
H
B
L
F
D

Node 5
E
P
I
A
M
C
G
N

Node 6
H
C
B
O
N
E
J
L
No vnodes restoration after node failure.
Token assignment & Vnodes

- With vnodes restoration after node failure.
Replication

- The Replication Factor
- Snitches
- Replication strategy
- Failure handling
- Repairing data
- Consistency Conflicts
- Consistency Levels
The Replication Factor

- For each keyspace (database), we define a `replication_factor`.

- Data is partitioned to `replication_factor` nodes - if set to 3, data will be located in 3 different nodes.

- All nodes are peers - no such term as main vs. replica.
Snitch

- The way Cassandra understands the network’s topology.
- Uses `endpoint_snitch` property in ‘cassandra.yaml’.
- Cassandra uses the snitch in order to route client’s requests to the nearest node to reduce network latency.
Locate the data in the node that is determined by the partitioner and then, in clockwise order, locate in next nodes.

```
CREATE KEYSPACE AddressBook
    WITH REPLICAATION = {
        'class' : 'SimpleStrategy',
        'replication_factor' : 3
    };
```
Replication Strategy - NetworkTopologyStrategy

Takes into consideration the network topology.

For example:

- Rack awareness - Locate the replicas in different racks.
- Configurable snitch - Let the user decide the replicas’ location.

```sql
CREATE KEYSPACE AddressBook
    WITH REPLICATION = {
        'class' : 'NetworkTopologyStrategy',
        'dc1' : 3,
        'dc2' : 2
    };
```
Repairing Data

- General idea - compare every piece of data and use the latest version.

- Three ways to repair data:
  - Synchronous read repair - on read operation, the coordinator node runs a checksum with all replicas and if necessary, the latest full replica is sent and matched.
  - Asynchronous read repair - each column family has an attribute called `read_repair_chance` that indicates the percentage of the time that will be spent to repair it.
  - Manually running repair - using nodetool repair.

`gc_grace_seconds` indicates the time between these automatic action that ran by the system.
Failure handling

- How to overcome node failure? (or - how to maintain the replication factor on error?)

- We spoke about the vnodes approach.

- Hinted handoff mechanism - by example:
  - Request has come to one node (called the coordinator) and one of the replicas isn’t reachable during write.
  - The system drops a hint on the coordinator node.
  - The hint includes the data itself along with metadata (location in cluster etc.)
  - Via gossip protocol, the coordinator will eventually get a signal that the node is back and will resend the write request to the node.
Failure handling

- By default, hints are stores for three hours to avoid long queues.
- After that time - run nodetool repair.
Cassandra offers a wide range of consistency levels for both read and write operations.

Using these terms:
- Strong consistency - The user will get the latest version of the data even if that means that he will be blocked until Cassandra resolves conflicts (recall the availability-consistency trade-off).
- Eventual consistency - The available data will be returned, even if that means that it is not the latest version available in the distributed keyspace.

Cassandra resolves conflict on requests or on “nodetool repair”.
Consistency Conflicts - How is it done?

- Every column has three fields:
  - key.
  - value.
  - timestamp.

- Follow the semantic of last-write-wins.

- Cassandra offers the opportunity to use QUORUM vote, using different scopes (to be discussed).
Consistency Levels - Examples

- **ANY**
  - Read - not permitted.
  - Write - data has to be written to commitlog and memtable of all replicas in the cluster.

- **ONE:**
  - Read - the replica from the closest node to the coordinator will be returned.
  - Write - data must be written to commitlog and memtable of at least one node. Other nodes will be repaired via the hinted handoff protocol.

- **LOCAL_ONE:**
  - Like ONE, but refers only to nodes in the same data-center.
Consistency Levels - Examples

- **QUORUM:**
  - Read - returns a record after a quorum vote from all data center was considered.
  - Write - data must be written to commitlog and memtable of a quorum replicas out of all data centers.

- **LOCAL QUORUM:**
  - Like QUORUM, but refers only to nodes in the same data-center.
We can combine methods to gain good consistency:

- **All and ONE:**
  - Use ALL for write will allow us to use ONE for reads.
  - The downside - if one of the nodes fell, the write will result with an exception called "UnavailableException".

- **QUORUM:**
  - Use QUORUM for both read and write.
  - Result is strong consistency.
Consistency Levels - Examples

- Highest consistency, Lowest availability
- Lowest consistency, Highest availability
Data Centers

- Use Cases
- Consistency across Data Centers
Use Cases (or - why do we need Data-Centers?)

- Live backups - use a second data-center with $replication\_factor = 1$ for backing up the database (like traditional databases).

- Failover - use a second data-center with exactly the same $replication\_factor$ and in case of failure, flip the databases.

- OLAP operations - run analytic operations in batch mode. After populating of the data, a repair should be ran to balance the two data-centers.
Use Cases (or - why do we need Data-Centers?)

Geographical span -

- Users from the same country usually interact more often.
- Improve latency.
- Allows the use of LOCAL_QUORUM.
- Requires repairs for consistency.
Consider the following scenario using LOCAL_QUORUM:

- Client sends write request.
- Coordinator node finds the relevant nodes using DHT and sends the request to the local nodes + one node in each remote data-center.
- Coordinator only waits until acknowledge received from local quorum, while remote nodes aren’t checked.
- We got a scenario in which we lost consistency.
Consistency across Data Centers

- **Solutions:**
  - Use non-local consistency level.
  - Run often nodetool repair.
Data Model

- RDBMS vs Cassandra
- Keyspaces
- Column
- Column Family
- Overview Diagram
For people coming from traditional RDBMs, the Cassandra data model can be strange and maybe even a bit confusing.

There are some terms such as keyspace completely new in Cassandra and some terms such as column does not match in the meaning in the RDBMs.

So before we dig into the data model of Cassandra we will illustrate how Cassandra can be mapped to RDBMs.
## Relational Model vs. Cassandra Model

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Cassandra Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Keyspace</td>
</tr>
<tr>
<td>Table</td>
<td>Column Family</td>
</tr>
<tr>
<td>Primary Key</td>
<td>Row Key</td>
</tr>
<tr>
<td>Column name</td>
<td>Column name</td>
</tr>
<tr>
<td>Column value</td>
<td>Column value</td>
</tr>
</tbody>
</table>
Keyspace

The outermost container for data in Cassandra. It is the schema and the unit for Cassandra’s access control mechanism.

- A cluster can hold several key spaces.
- When you create key space, there are set of attributes you can set:
  - Replication Factor: The number of nodes holding the replica of data.
  - Replica Placement Strategy: The strategy the replication should happen.
  - Column Families: a Keyspace can define many column families.
The basic building block. A column is a triplet:

- Column name.
- Column value.
- Timestamp. - is used for internal Cassandra use and is not accessible or editable by the use.
Here is an example of a column, represented with JSON notation,

```json
{
  "name": "email",
  "value": "joe@gmail.com",
  "timestamp": 123456789234
}
```
Column Family

A column family is a container for an ordered collection of rows, each of which is itself an ordered collection of columns.

A column family will be number of rows and each row can have variable length of columns.
CQL

- CQL Data Representation
- Single Primary Keys
- Compound Keys
- Partition Keys
- Clustering Columns
- Composite Partition Keys
- Queries
CQL Data Representation

CQL is intended to provide a common, simpler and easier to use interface into Cassandra.

The CQL data representation does not always match the underlying storage structure. However, it introduces an abstraction on top of the storage rows.

Keep in mind that CQL is not SQL, so you must understand what’s happening under the covers.
Single Primary Key

Our first model is very basic. We define a table which called books with a single primary key, title.

```
CREATE TABLE books (  
  title text,  
  author text,  
  year int,  
  PRIMARY KEY (title)  
);
```
Insert data to books table

```
INSERT INTO books (title, author, year)
VALUES ('Patriot Games', 'Tom Clancy', 1987);

INSERT INTO books (title, author, year)
VALUES ('Without Remorse', 'Tom Clancy', 1993);
```

We insert some data. In this case we add 2 rows.
Read our newly inserted rows

```
SELECT * FROM books;
```

<table>
<thead>
<tr>
<th>title</th>
<th>author</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without Remorse</td>
<td>Tom Clancy</td>
<td>1993</td>
</tr>
<tr>
<td>Patriot Games</td>
<td>Tom Clancy</td>
<td>1987</td>
</tr>
</tbody>
</table>
Under the hood: Looks different from SQL

RowKey: Without Remorse
=> (name=, value=, timestamp=1393102991499000)
=> (name=author, value=Tom Clancy, timestamp=1393102991499000)
=> (name=year, value=1993, timestamp=1393102991499000)

RowKey: Patriot Games
=> (name=, value=, timestamp=1393102991499100)
=> (name=author, value=Tom Clancy, timestamp=1393102991499100)
=> (name=year, value=1987, timestamp=1393102991499100)
Important Features

Row key is distributed randomly using a hash algorithm, so the results are returned in no particular order.

By contrast, columns are sorted in sorted order by name. “author” comes before “year”.

These are critical as they are central to effective data modeling.
### Create Table: authors

- **Table Schema:**
  - `name` (text)
  - `year` (int)
  - `title` (text)
  - `isbn` (text)
  - `publisher` (text)

- **Primary Key:** `(name, year, title)`

#### Compound Keys

<table>
<thead>
<tr>
<th>name</th>
<th>year</th>
<th>title</th>
<th>isbn</th>
<th>publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom Clancy</td>
<td>1987</td>
<td>Patriot Games</td>
<td>0-399-1234-4</td>
<td>Putnam</td>
</tr>
<tr>
<td>Tom Clancy</td>
<td>1993</td>
<td>Without Remorse</td>
<td>0-399-1456-0</td>
<td>Putnam</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Partition Key:** `name`
- **Clustering Columns:** `year, title`
Partition Keys

When declaring a primary key, the first field in the list is always the partition key.

This translates directly to the storage row key, which is randomly distributed in the cluster via hashing.

That is crucial since in general, you must provide the partition key when issuing a query.
Clustering Columns

The remaining fields in the primary key.

These determine the ordering of the data on disk.

They play a key role in determining the kinds of queries you can run against your data.
So how it looks like

<table>
<thead>
<tr>
<th>RowKey: Tom Clancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>=&gt; (name=1987:Patriot Games:ISBN, value=0-399-13241-4)</td>
</tr>
<tr>
<td>=&gt; (name=1987:Patriot Games:publisher, value=Putnam)</td>
</tr>
<tr>
<td>=&gt; (name=1993:Without Remorse:ISBN, value=0-399-13825-0)</td>
</tr>
<tr>
<td>=&gt; (name=1993:Without Remorse:publisher, value=Putnam)</td>
</tr>
</tbody>
</table>
It is also possible to reverse the stored sort order.

```sql
CREATE TABLE authors (  
  name text,  
  year int,  
  title text,  
  isbn text,  
  publisher text,  
  PRIMARY KEY (name, year, title)  
) WITH CLUSTERING ORDER BY (year DESC);
```
CREATE TABLE authors ( 
    name text, 
    year int, 
    title text, 
    isbn text, 
    publisher text, 
    PRIMARY KEY ((name, year), title)
);
RowKey: Tom Clancy:1993
=> (name=Without Remorse:isbn, value=0-399-13241-4)
=> (name=Without Remorse:publisher, value=5075746e616d)

RowKey: Tom Clancy:1987
=> (name=Patriot Games:isbn, value=0-399-13825-0)
=> (name=Patriot Games:publisher, value=5075746e616d)
It matters how the data is stored

- Queries must respect the underlying storage.
- Partition keys must be chosen carefully.
- Cassandra handles range queries very well.
- Carefully order your clustering columns.
SELECT * FROM authors WHERE name='Tom Clancy';

RowKey: Tom Clancy

=> (name=1996:Executive Orders:publisher, value=Putnam)

=> (name=1996:Executive Orders:ISBN, value=0-399-13825-0)

=> (name=1994:Debt of Honor:publisher, value=Putnam)

...
Range Queries

```
SELECT * FROM authors WHERE name='Tom Clancy' AND year >= 1993;
```

RowKey: Tom Clancy

=> (name=1996:Executive Orders:publisher, value=Putnam)

=> (name=1996:Executive Orders:ISBN, value=0-399-13825-0)

=> (name=1994:Debt of Honor:publisher, value=Putnam)


=> (name=1993:Without Remorse:publisher, value=Putnam)

=> (name=1993:Without Remorse:ISBN, value=0-399-13825-0)
Failed Queries

SELECT * FROM authors WHERE publisher = 'Putnam';

Bad Request: No indexed columns present in by-columns clause with Equal operator

Solution : Secondary Indices
Special Features

- Collections
- List Example
- Expiring Columns
Collections

They add richer capabilities that give developers more flexibility when modeling certain types of data.

Cassandra supports three collection types: sets, lists, and maps.
CREATE TABLE authors (  
    name text,  
    books list<text>,  
    PRIMARY KEY (name) 
);
List Example (cont’): Insert

```
INSERT INTO authors (name, books)
VALUES ('Tom Clancy', ['Without Remorse', 'Patriot Games']);
```
List Example (cont'): Update

UPDATE authors
SET books = books - ['Red Storm Rising']
WHERE name = 'Tom Clancy';

UPDATE authors
SET books = ['Red Storm Rising'] + books
WHERE name = 'Tom Clancy';
List Example (cont’): Storage Layer

RowKey: Tom Clancy

=> (name=books:d36de8b0305011e4a0dddbbeade718be, value=576974686f)

=> (name=books:d36de8b1305011e4a0dddbbeade718be, value=506174726)
INSERT INTO authors (name, title)
VALUES ('Tom Clancy', 'Patriot Games')
USING TTL 86400;

UPDATE authors USING TTL 86400
SET publisher = 'Putnam'
WHERE name = 'Tom Clancy'
AND title = 'Patriot Games'
AND year = 1987;
No JOIN

Because if its distributed nature, Cassandra has no support for RDBMS style joins.

Solutions are:

Perform separate queries and then have your application join the data itself.

Denormalize the data and store a materialized view of the join.(Chapter 8)
Summary - What does Cassandra provide?

- Open-Source
- Peer to Peer Architecture
- Elastic Scalability
- High Availability & Fault Tolerance
- High Performance
- Column Oriented
- Tunable Consistency
- Schema-Free