NOTES:

1. Many thanks to DataStax for providing this material and granting us permission to use it!

2. These slides have been lightly modified for CS122D use.
Apache Cassandra™ vs. SQL
Apache Cassandra™ vs. Relational (SQL)

Simple differentiation overview

<table>
<thead>
<tr>
<th>Relational Database</th>
<th>Apache Cassandra™</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample relational model methodology</td>
<td>Cassandra modeling methodology</td>
</tr>
<tr>
<td>Data—Model—Application</td>
<td>Application—Model—Data</td>
</tr>
<tr>
<td>Entities are king</td>
<td>Queries are king</td>
</tr>
<tr>
<td>Primary key for uniqueness</td>
<td>Primary keys are much more</td>
</tr>
<tr>
<td>Often have single point of failure</td>
<td>Distributed architecture</td>
</tr>
<tr>
<td>ACID compliant</td>
<td>CAP theorem</td>
</tr>
<tr>
<td>Joins and indexes</td>
<td>Denormalization</td>
</tr>
<tr>
<td>Referential Integrity enforced</td>
<td>RI not enforced</td>
</tr>
</tbody>
</table>
Data Modeling Methodologies

Relational vs. Apache Cassandra™

```
<table>
<thead>
<tr>
<th>Data</th>
<th>Model</th>
<th>Application</th>
</tr>
</thead>
</table>
```

```
| Application | Model | Data |
```

© Copyright 2019. Use only with permission.
Relational Data Modeling

Sample Methodology—One of many!

CDM: Conceptual data model
LDM: Logical data model
PDM: Physical data model
Cassandra Data Modeling

Cassandra’s data modeling methodology—We will explore fully later!
Transactions in Apache Cassandra™

Cassandra does not support ACID transactions

- ACID causes a significant performance penalty
- Not required for many use cases
- However, a single Cassandra write operation demonstrates ACID properties
  - INSERTs, UPDATEs, and DELETEs are atomic, isolated, and durable
  - Tunable consistency for data replicated to nodes, but does not handle application integrity constraints
Apache Cassandra™ and CAP Theorem

Consistency, Availability, and Partition Tolerance

- By default, Cassandra is an AP database
- However, this is tunable with consistency level
- By tuning consistency level, you can make it more CP than AP
- Note that Cassandra isn’t designed to be CA because you can’t sacrifice partition tolerance
Apache Cassandra™ and Denormalization

Cassandra doesn’t support joins

```
CREATE TABLE comments_by_video (    
    video_title text,
    comment_id timeuuid,
    user_id text,
    video_id timeuuid,
    comment text,
    PRIMARY KEY ((video_title), comment_id)
);
```

```
CREATE TABLE comments_by_user (    
    user_login text,
    comment_id timeuuid,
    user_id text,
    video_id timeuuid,
    comment text,
    PRIMARY KEY ((user_login), comment_id)
);
```
KillrVideo: A Use Case

KillrVideo Features

- Videos
- User Accounts
- User Ratings
- Movies/TV Shows
- Search
- Playback
- Comments
Problems KillrVideo Faces

- **Scalability**—Must be able to support constant addition of new users and videos
- **Reliability**—Must always be available
- **Ease of use**—Must be easy to manage and maintain

- Single points of failure
- Scaling complexity
- Reliability issues
- Difficult to serve users worldwide
KillrVideo on DataStax Enterprise

- Peers instead of master/slave
- Linear scale performance
- Always-on reliability
- Data can be stored geographically close to clients
Apache Cassandra™ Tables
Apache Cassandra™ Terminology

Terms and definitions to get your head around

- **Data model:**
  - An abstract model for organizing elements of data
  - In Apache Cassandra™ this is based on the **queries** you want to perform

- **Keyspace:**
  - Similar to a relational schema—outermost grouping of data
  - All tables live inside a keyspace
  - Keyspace is also the container for **replication**

- **Table:**
  - Grouped into keyspaces
  - Contain columns

- **Partition:**
  - Row(s) of data that are stored on a particular node in your cluster based on a partitioning strategy
Apache Cassandra™ Table Terminology

More specific to the tables themselves

- **Row:**
  - One or more CQL rows stored together on a partition

- **Column:**
  - Similar to a column in a relational database

- **Primary key:**
  - Used to access the data in a table and guarantees uniqueness

- **Partition key:**
  - Defines the node on which the data is stored

- **Clustering column:**
  - Defines the order of rows within a partition
Graphical Representation of Terms

A picture is worth 1000 words
Data Types—Integers

Integer data types

- TINYINT: 8-bit signed integer
- SMALLINT: 16-bit signed
- INT: 32-bit signed integer
- BIGINT: 64-bit signed integer
- VARINT: Arbitrary-precision integer--F-8 encoded string
- DECIMAL: Variable-precision decimal, supports integers and floats.
- FLOAT: 32-bit IEEE-754 floating point
- DOUBLE: 64-bit IEEE-754 floating point
Apache Cassandra™ Data Types—Text

Text data types

- **ASCII**: US-ASCII characters
- **TEXT**: UTF-8 encoded string
- **VARCHAR**: UTF-8 encoded string
Data Types—Date, Time and Unique Identifiers

Time, timestamp and unique identifiers

- **DATE**: 32-bit unsigned integer—number of days since epoch (Jan 1, 1970)
- **DURATION**: Signed 64-bit integer—amount of time in nanoseconds
- **TIME**: Encoded 64-bit signed—number of nanoseconds since midnight
- **TIMESTAMP**: 64-bit signed integer—date and time since epoch in milliseconds

- **UUID**: 128 bit universally unique identifier—generate with the `UUID()` function
- **TIMEUUID**: unique identifier that includes a “conflict-free” timestamp—generate with the `NOW()` function
Data Types—Specialty Types

Some that don't fit the other categories

• **BOOLEAN**: Stored internally as *true* or *false*

• **BLOB**: Arbitrary bytes (no validation), expressed as hexadecimal

• **INET**: IP address string in IPv4 or IPv6 format

• **COUNTER**: 64-bit signed integer—only one counter column is allowed per table
Data Partitioning and Storage Structure
Partitions

To truly understand Cassandra data modeling, you must master its partitioning concepts!

Key concepts (so to speak 😊) include:

- Partitions
- Partition keys
- Composite partition keys
- Clustering keys
Queries in a relational context

In a relational world, these queries should work on the videos table

```
SELECT *
FROM videos
WHERE name = 'Frozen';

SELECT name, added_date
FROM videos
WHERE added_date >= '2013-11-25';
```

Let’s go try them on Apache Cassandra™ ...

https://astra.datastax.com/

```
SELECT * FROM videos WHERE name = 'Frozen';
SELECT name, added_date FROM videos WHERE added_date >= '2013-11-25' LIMIT 5;
SELECT name, added_date, tags FROM videos WHERE tags CONTAINS 'walt disney';
DESCRIBE videos; -- notice the key and also the 1NF relaxation
```
Videos Table Example

Simpler videos schema for illustration

```
CREATE TABLE videos (  
id int,  
name text,  
runtime int,  
year int,  
PRIMARY KEY (id)  
);
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>runtime</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insurgent</td>
<td>119</td>
<td>2015</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
</tr>
<tr>
<td>3</td>
<td>Mockingjay</td>
<td>122</td>
<td>2014</td>
</tr>
</tbody>
</table>
Apache Cassandra™ Storage Structure

Partitions

Partition Key

Cells

Key

Value

1
- name: Insurgent
- runtime: 144
- year: 2015

2
- name: Interstellar
- runtime: 98
- year: 2014

3
- name: Mockingjay
- runtime: 122
- year: 2014
Queries + Partitions + Keys + Storage = ?

**WHERE on non-primary key columns**

- Apache Cassandra™ distributes partitions across **nodes**
- WHERE on any field other than partition key would require a scan of all partitions on all nodes
- Intra-node efficiency is also a considered concern
- *Just say “no” to inefficient access patterns...!*
CQL Tables

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>runtime</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insurgent</td>
<td>119</td>
<td>2015</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
</tr>
<tr>
<td>3</td>
<td>Mockingjay</td>
<td>122</td>
<td>2014</td>
</tr>
</tbody>
</table>

SELECT * FROM videos
Determining Partition Keys

CQL's PARTITION KEY clause determines partitioning criteria in the primary key

```
PRIMARY KEY (id)
```

1
- name: Insurgent
- runtime: 144
- year: 2015

2
- name: Interstellar
- runtime: 98
- year: 2014

3
- name: Mockingjay
- runtime: 122
- year: 2014
Primary Key

Simple Primary Key

- Contains only the partition key
- Determines which node stores the data
- Must (of course!) be unique

<table>
<thead>
<tr>
<th>User</th>
<th>Email</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a7e78478-0a54-4949-90f3-14ec4cbea40c</td>
<td><a href="mailto:jbellis@datastax.com">jbellis@datastax.com</a></td>
<td>Jonathan</td>
</tr>
<tr>
<td>67657da3-4443-46ab-b60a-510a658fc7bb</td>
<td><a href="mailto:matt@datastax.com">matt@datastax.com</a></td>
<td>Matt</td>
</tr>
<tr>
<td>3b1f62b1-386b-46e3-b55d-00f1abbaf8b2b</td>
<td><a href="mailto:patrick@datastax.com">patrick@datastax.com</a></td>
<td>Patrick</td>
</tr>
</tbody>
</table>
Composite Partition Keys

Multi-value primary key

CREATE TABLE videos (  
    name text,  
    runtime int,  
    year int,  
    PRIMARY KEY ((name, year))  
);
Primary Key vs. Partition Key

Wait, what? Remind me again what these are?

• **Partition key**: The part of the primary key that determines what node the partition is stored on.

• **Primary key**: Includes partition key and any/all clustering columns – and again, unique per row.

• Can these be the same? Yes! But not usually.

• Proper **key design** is “key” in Cassandra-land!
Clustering Columns

How Cassandra sorts data within each partition

• Appear after partition key within PRIMARY KEY clause
• Data displays the same as before

```
CREATE TABLE videos (
    id int,
    name text,
    runtime int,
    year int,
    PRIMARY KEY ((year), name)
);
```

<table>
<thead>
<tr>
<th>year</th>
<th>name</th>
<th>id</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>Insurgent</td>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>2014</td>
<td>Interstellar</td>
<td>2</td>
<td>98</td>
</tr>
<tr>
<td>2014</td>
<td>Mockingjay</td>
<td>3</td>
<td>122</td>
</tr>
</tbody>
</table>
Clustering Columns, Cont.

Clustering sorts CQL rows in partitions

<table>
<thead>
<tr>
<th>year</th>
<th>name</th>
<th>id</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015</td>
<td>Insurgent</td>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>2014</td>
<td>Interstellar</td>
<td>2</td>
<td>98</td>
</tr>
<tr>
<td>2014</td>
<td>Mockingjay</td>
<td>3</td>
<td>122</td>
</tr>
</tbody>
</table>

© Copyright 2019. Use only with permission.
Side-by-Side Comparison

Primary key with just ID vs. with year and title.

<table>
<thead>
<tr>
<th>PRIMARY KEY((id))</th>
<th>PRIMARY KEY((year), title)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>runtime</td>
</tr>
<tr>
<td>Insurgent</td>
<td>144</td>
</tr>
<tr>
<td>year</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>runtime</td>
</tr>
<tr>
<td>Interstellar</td>
<td>98</td>
</tr>
<tr>
<td>year</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>runtime</td>
</tr>
<tr>
<td>Mockingjay</td>
<td>122</td>
</tr>
<tr>
<td>year</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>Interstellar: id</td>
<td>2</td>
</tr>
<tr>
<td>Interstellar: runtime</td>
<td>98</td>
</tr>
<tr>
<td>Mockingjay: id</td>
<td>3</td>
</tr>
<tr>
<td>Mockingjay: runtime</td>
<td>113</td>
</tr>
<tr>
<td>2015</td>
<td></td>
</tr>
<tr>
<td>Insurgent: id</td>
<td>1</td>
</tr>
<tr>
<td>Insurgent: runtime</td>
<td>119</td>
</tr>
</tbody>
</table>

© Copyright 2019. Use only with permission.
Cluster Column Ordering

Clustering column values stored sorted

Interstellar then Mockingjay (ascending)
Cluster Column Ordering—Descending

Default is ascending (but you can specify descending)

Interstellar then Mockingjay (ascending)
Cluster Column Ordering

WITH CLUSTERING ORDER BY

CREATE TABLE videos (  
id int,  
name text,  
runtime int,  
year int,  
PRIMARY KEY ((year), name)  
) WITH CLUSTERING ORDER BY (name DESC);
Querying Clustering Columns

You can query on clustering columns because lookup is fast

```
SELECT *
FROM videos
WHERE year = 2014 AND name = 'Mockingjay';
```
Querying Clustering Columns—Range

You can also do a range query on clustering columns

```sql
SELECT *
FROM videos
WHERE year = 2014 AND name >= 'Interstellar';
```
Denormalization
## Typical Relational Structure

### Relational example *(data)*

<table>
<thead>
<tr>
<th>videos</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>title</td>
<td>runtime</td>
<td>year</td>
</tr>
<tr>
<td>1</td>
<td>Insurgent</td>
<td>119</td>
<td>2015</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
</tr>
<tr>
<td>3</td>
<td>Mockingjay</td>
<td>122</td>
<td>2014</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>users</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>login</td>
<td>name</td>
</tr>
<tr>
<td>a</td>
<td>emotions</td>
<td>Mr. Emotional</td>
</tr>
<tr>
<td>b</td>
<td>clueless</td>
<td>Mr. Naïve</td>
</tr>
<tr>
<td>c</td>
<td>noshow</td>
<td>Mr. Inactive</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>comments</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>user_id</td>
<td>video_id</td>
<td>comment</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
<td>Loved it!</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>3</td>
<td>Hated it!</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>2</td>
<td>I cried at the end!</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>2</td>
<td>Someone stole my tissues...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Query Comments By Video Title

Relational example (*join query*)

```
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```
Query Comments By Video Title, Cont.

```sql
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>runtime</th>
<th>year</th>
<th>id</th>
<th>user_id</th>
<th>video_id</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insurgent</td>
<td>119</td>
<td>2015</td>
<td>1</td>
<td>a</td>
<td>1</td>
<td>Loved it!</td>
</tr>
<tr>
<td>3</td>
<td>Mockingjay</td>
<td>122</td>
<td>2014</td>
<td>2</td>
<td>a</td>
<td>3</td>
<td>Hated it!</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
<td>3</td>
<td>a</td>
<td>2</td>
<td>I cried at the end!</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
<td>4</td>
<td>b</td>
<td>2</td>
<td>Someone stole my tissues...</td>
</tr>
</tbody>
</table>

...
Query Comments By Video Title, Cont.

WHERE title = 'Interstellar'

```
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>runtime</th>
<th>year</th>
<th>id</th>
<th>user_id</th>
<th>video_id</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insurgent</td>
<td>119</td>
<td>2015</td>
<td>1</td>
<td>a</td>
<td>1</td>
<td>Loved it!</td>
</tr>
<tr>
<td>3</td>
<td>Mockingjay</td>
<td>122</td>
<td>2014</td>
<td>2</td>
<td>a</td>
<td>3</td>
<td>Hated it!</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
<td>3</td>
<td>a</td>
<td>2</td>
<td>I cried at the end!</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
<td>4</td>
<td>b</td>
<td>2</td>
<td>Someone stole my tissues...</td>
</tr>
</tbody>
</table>
Query Comments By Video Title, Cont.

Results of the query

```
SELECT comment
FROM videos
JOIN comments
ON videos.id = comments.video_id
WHERE title = 'Interstellar'
```

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>runtime</th>
<th>year</th>
<th>id</th>
<th>user_id</th>
<th>video_id</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
<td>3</td>
<td>a</td>
<td>2</td>
<td>I cried at the end!</td>
</tr>
<tr>
<td>2</td>
<td>Interstellar</td>
<td>98</td>
<td>2014</td>
<td>4</td>
<td>b</td>
<td>2</td>
<td>Someone stole my tissues...</td>
</tr>
</tbody>
</table>
Query Comments By User Login

```sql
SELECT comment
FROM users
JOIN comments
ON users.id = comments.user_id
WHERE user.login = 'emotions'
```
Query Comments By User Login, Cont.

```sql
SELECT comment
FROM users
JOIN comments
ON users.id = comments.user_id
WHERE user.login = 'emotions'
```

<table>
<thead>
<tr>
<th>users</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>login</td>
</tr>
<tr>
<td>a</td>
<td>emotions</td>
</tr>
<tr>
<td>b</td>
<td>clueless</td>
</tr>
<tr>
<td>c</td>
<td>noshow</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

© Copyright 2019. Use only with permission.
Denormalizing For Query Performance

This is how Apache Cassandra™ works around having no joins

**comments_by_video**

- **Insurgent**
  - 1:comment: Loved it!
  - 1:user_id: a
  - 1:video_id: 1

- **Mockingjay**
  - 2:comment: Hated it!
  - 2:user_id: a
  - 2:video_id: 3

- **Interstellar**
  - 3:comment: I cried at the end!
  - 3:user_id: a
  - 3:video_id: 2
  - 4:comment: Someone stole...
  - 4:user_id: b
  - 4:video_id: 2

**comments_by_user**

- **emotions**
  - 1:comment: Loved it!
  - 1:user_id: a
  - 1:video_id: 1
  - 2:comment: Hated it!
  - 2:user_id: a
  - 2:video_id: 3
  - 3:comment: I cried at the end!
  - 3:user_id: a
  - 3:video_id: 2

- **clueless**
  - 4:comment: Someone stole...
  - 4:user_id: b
  - 4:video_id: 2

© Copyright 2019. Use only with permission.
Denormalizing For Query Performance

CREATE TABLE comments_by_video (
    video_title text,
    comment_id timeuuid,
    user_id text,
    video_id timeuuid,
    comment text,
    PRIMARY KEY ((video_title), comment_id)
);

CREATE TABLE comments_by_user (
    user_login text,
    comment_id timeuuid,
    user_id text,
    video_id timeuuid,
    comment text,
    PRIMARY KEY ((user_login), comment_id)
);
Denormalizing For Query Performance

The other critical Cassandra database design puzzle piece (along with key design)
To Be Continued....