Introduction to HBASE

Advanced Databases

Enric Biosca Trias ebiosca@maia.ub.es

Dept. Matemàtica Aplicada i Anàlisi.

Universitat de Barcelona

Chapter Topics

Introduct to HBase

- What is HBase?
- Strengths of HBase
- HBase in Production
- Weaknesses of HBase

What is HBase?

- **HBase is a NoSQL database that runs on top of HDFS**
- **HBase is…**
  - Highly available and fault tolerant
  - Very scalable, and can handle high throughput
  - Able to handle massive tables
  - Well suited to sparse rows where the number of columns varies
  - An open-source, Apache project
- **HDFS provides:**
  - Fault tolerance
  - Scalability

Google BigTable

- **HBase is based on Google's BigTable**
- **The Google use case is to search the entire Internet**
  - BigTable is at the center of how the company accomplishes this
- **Engineering considerations for searching the Internet:**
  - How do you efficiently download the Web pages?
  - How do you store the entire Internet?
  - How do you index the Web pages?
  - How do you efficiently search the Web pages?

Hbase Principal value?

- **HBase helps solve data access issues where random access is required**
- **HBase scales easily, making it ideal for Big Data storage and processing needs**
- **Columns in an HBase table are defined dynamically, as required**

HBase Usage Scenarios (1)

- **High capacity**
  - Massive amounts of data
    - Hundreds of gigabytes, growing to terabytes or even petabytes
  - Example: Storing the entire Internet
- **High read and write throughput**
  - 1000s/second per node
  - Example: Facebook Messages
    - 75 billion operations per day
    - Peak usage of 1.5 million operations per second
    - Runs entirely on HBase

HBase Usage Scenarios (2)

- **Scalable in-memory caching**
  - Adding nodes adds to available cache
- **Large amount of stored data, but queries o`en access a small subset**
  - Data is cached in memory to speed up queries by reducing disk I/O
- **Data layout**
  - HBase excels at key lookup
  - No penalty for sparse columns

When To Use HBase

- **Use HBase if…**
  - You need random write, random read, or both (but not neither)
  - Your application performs thousands of operations per second on multiple terabytes of data
  - Your access patterns are well-known and relatively simple
- **Don't use HBase if…**
  - Your application only appends to your dataset, and tends to read the whole thing when processing
  - Primary usage is for ad-hoc analytics (non-deterministic access patterns)
  - Your data easily fits on one large node

HBase In Production

- **Many enterprises are using HBase in production**
  - Many use cases are mission critical
- **Examples of companies using HBase:**
  - eBay
  - Facebook
  - FINRA
  - Pinterest
  - Salesforce
  - StumbleUpon
  - TrendMicro
  - TwiCer
  - Yahoo!
  - More complete list at http://wiki.apache.org/ hadoop/ Hbase/ PoweredBy

HBase Use Cases

Messaging

- Facebook Messages
- Telco SMS/MMS services
- Feeds like Tumblr, Pinterest

Simple Entities

- Geolocation data
- Search index building

Graph Data

- Sessionization
  - Financial transactions
  - Click streams
  - Network traffic

Metrics

- Campaign impressions
- Click counts
- Sensor Data

UNIVERSITAT DE BARCELONA

HBase Use Case Characteristics

Messaging

- Facebook Messages
- Telco SMS/MMS services
- Feeds like Tumblr, Pinterest

Characteris-cs

- Real time random writes
- Reading of top N entries

Simple Entities

- Geolocation data
- Search index building

Characteristics

- Batch or real time, random writes
- Real-time, random reads

Graph Data

- Sessionization
  - Financial transactions
  - Click streams
  - Network traffic

Characteristics

- Batch/real time, random reads/writes

Metrics

- Campaign impressions
- Click counts
- Sensor Data

Characteristics

- Frequently updated metrics

Features Missing from HBase

- **HBase does not provide certain popular RDBMS features**
  - Integrated support for SQL
    - External projects such as Hive, Impala, and Phoenix provide various ways of using SQL to access HBase tables
  - Support for transactions
  - Multiple indexes on a table
  - ACID compliance

Different Design Approach in HBase

- **Designing an HBase application requires developers to engineer the system using a data-centric approach**
  - This is a less familiar approach; relationship-centric is more common
  - We will cover this in detail later in the course

Essential Points

- There are many technological challenges when dealing with Big Data
- Hadoop is a distributed system aimed at managing Big Data
- Hadoop's core components are HDFS and MapReduce
- HBase is a NoSQL database that runs on top of HDFS
- HBase is not a tradi- onal RDBMS
- HBase requires a data-centric design approach
- HBase allows for large tables and high throughput

HBASE TABLES

Chapter Topics

HBase Tables

- **HBase Concepts**
- HBase Table Fundamentals
- Thinking About Table Design
- Hands-On Exercise: HBase Data Import
- Essential Points

HBase Terms

- **Definitions**
  - *Node*
    - A single computer
  - *Cluster*
    - A group of nodes connected and coordinated by certain nodes to perform tasks
  - *Master Node*
    - A node performing coordination tasks
  - *Worker Node*
    - A node performing tasks assigned to it by a master node
  - *Daemon*
    - A process or program that runs in the background

Fundamental HBase Concepts

- **HBase stores data in tables**
  - Similar to RDBMS tables but with some important differences
- **Table data is stored on the Hadoop Distributed File System (HDFS)**
  - Data is split into HDFS blocks and stored on multiple nodes in the cluster

What is a Table?

- HBase tables are comprised of rows, columns, and column families
- Every row has a *row key* for fast lookup
- Columns hold the data for the table
- Each column belongs to a parCcular *column family*
- A table has one or more column families

About HBase Tables

- **HBase is essentially a distributed, sorted map**
- **Distributed: HBase is designed to use multiple machines to store and serve table data**
- **Sorted Map: HBase stores table data as a map, and guarantees that adjacent keys will be stored next to each other on disk**

Example Application Data (1)

- **In our examples, we will use a table that holds**
  - User contact information
  - Profile photos
  - User sign-in information such as username and password
  - Settings or preferences for multiple applications
- **The table will be designed to provide access to the data based on the username**

Example Application Data (2)

- **Not every field will have a value**
  - This might happen if the application that stores data to a given field has not run
  - Alternatively, the user may have elected not to provide all the information
- **For now we will focus on the contact information and profile photo**
  - Contact information
    - First Name
    - Last Name
  - Profile photo
    - Image that the user uploads

Rows

- **Tables are comprised of rows, columns, and column families**
- **Every row has a *row key***
  - A row key is analogous to a primary key in a traditional RDBMS
  - Rows are stored sorted by row key to enable speedy retrieval of data

Columns

- **Columns hold the data for the table**
  - Columns can be created on the fly
    - A column exists for a particular row only if the row has data in that column
  - The table's *cells* (row-column intersection) are arbitrary arrays of bytes
- **Each column in an HBase table belongs to a particular *column family***
  - A collection of columns
- **A table has one or more column families**
  - Created as part of the table definition

HBase Columns and Column Families

- **All columns belonging to the same column family have the same prefix**
  - e.g., **`contactinfo:fname`** and **`contactinfo:lname`**
  - The "**`:`**" delimits the column family from the qualifier (column name)
- **Tuning and storage settings can be specified for each column family**
  - For example, the number of versions of each cell which will be stored
  - More on this later
- **A column family can have any number of columns**
  - Columns within a family are sorted and stored together

HBase Tables: A Conceptual View

- **A column is referenced using its column family and column name (or *qualifier*)**
- **Separate column families are useful for**
  - Data that is not frequently accessed together
  - Data that uses different column family options, e.g., compression

Column Families

Column Names

Rows

	contactinfo		profilephoto
Row Key	fname	lname	image
jdupont	Jean	Dupont	
jsmith	John	Smith	<smith.jpg>
mrossi	Mario	Rossi	<mario.jpg>

Storing Data in Tables

- **Data in HBase tables is stored as byte arrays**
  - Anything that can be converted to an array of bytes can be stored
    - Strings, numbers, complex objects, images, etc.
- **Cell size**
  - Practical limit on the size of values
  - In general, cell size should not consistently be above 10MB

UNIVERSITAT DE BARCELONA

Table Storage On Disk: Basics

- **Data is physically stored on disk on a per-column family basis**
- **Empty cells are not stored**

File

	contactinfo	
Row Key	fname	lname
jdupont	Jean	Dupont
jsmith	John	Smith
mrossi	Mario	Rossi

File

	profilephoto
Row Key	image
jsmith	<smith.jpg>
mrossi	<mario.jpg>

Data Storage Within a Column Family

- **HBase table features**
  - Row key + column + timestamp --> value
  - Row key and value are just bytes
  - Can store anything that can be serialized into a byte array

Sorted by Row Key and Column

Row Key	Column	Timestamp	Cell Value
jdupont	contactinfo:fname	1273746289103	Jean
jdupont	contactinfo:lname	1273878447049	Dupont
jsmith	contactinfo:fname	1273516197868	John
jsmith	contactinfo:lname	1273871824184	Smith
mrossi	contactinfo:fname	1273616297446	Mario
mrossi	contactinfo:lname	1273971921442	Rossi

HBase Operations (1)

- **All rows in HBase are identified by a row key**
  - This is like the primary key in a relational database
- **`Get/Scan` retrieves data**
  - A **`Get`** retrieves a single row using the row key
  - A **`Scan`** retrieves all rows
  - A **`Scan`** can be constrained to retrieve all rows between a start row key and an end row key
- **`Put` inserts data**
  - A **`Put`** adds a new row identified by a row key
  - Multiple **`Put`** calls can be run to insert multiple rows with different row keys

HBase Operations (2)

- **`Delete` marks data as having been deleted**
  - A **`Delete`** removes the row identified by a row key
  - The data is not removed from HDFS during the call but is marked for deletion
  - Physical deletion from HDFS happens later
- **`Increment` allows atomic counters**
  - Cells containing a value stored as a 64-bit integer (a **`long`**)
  - **`Increment`** allows the value to be initially set, or incremented if it already has a value
  - Atomicity allows for concurrent access from multiple clients without fear of corruption by a write from another process

Row Key is the Only Indexed Column

- **RDBMSs can have as many index columns as required**
- **In HBase, we have just one indexed column – the row key**
- **Significant effort goes into the row key planning for HBase tables**
  - We rely on the row key to provide quick access to data for all applications that use a given table

Features Comparison: RDBMS vs. HBase

	RDBMS	HBase
Data layout	Row- or column-oriented	Column family-oriented
Transactions	Yes	Single row only
Query language	SQL	get/put/scan
Security	Authentication/ Authorization	Access control at per-cell level, also at cluster, table, or row level
Indexes	Yes	Row key only
Max data size	TBs	PB+
Read/write throughput limits	1000s queries/second	Millions of queries/second

Replacing RDBMSs with HBase

- **Replacing an RDBMS-based applicaCon with HBase requires significant re-architecting**
- **Some major differences**
  - Data layout
  - Data access

Comparison with RDBMS Table Design

- **Joins**
  - In relational databases, one would typically normalize tables and use joins to retrieve data
  - HBase does not support explicit joins
    - Instead, a lookup by row key joins data from column families
- **Scaling**
  - Relational tables can scale through partitioning or sharding data
  - HBase automatically partitions data into smaller pieces

HBase vs. RDBMS Schema Design

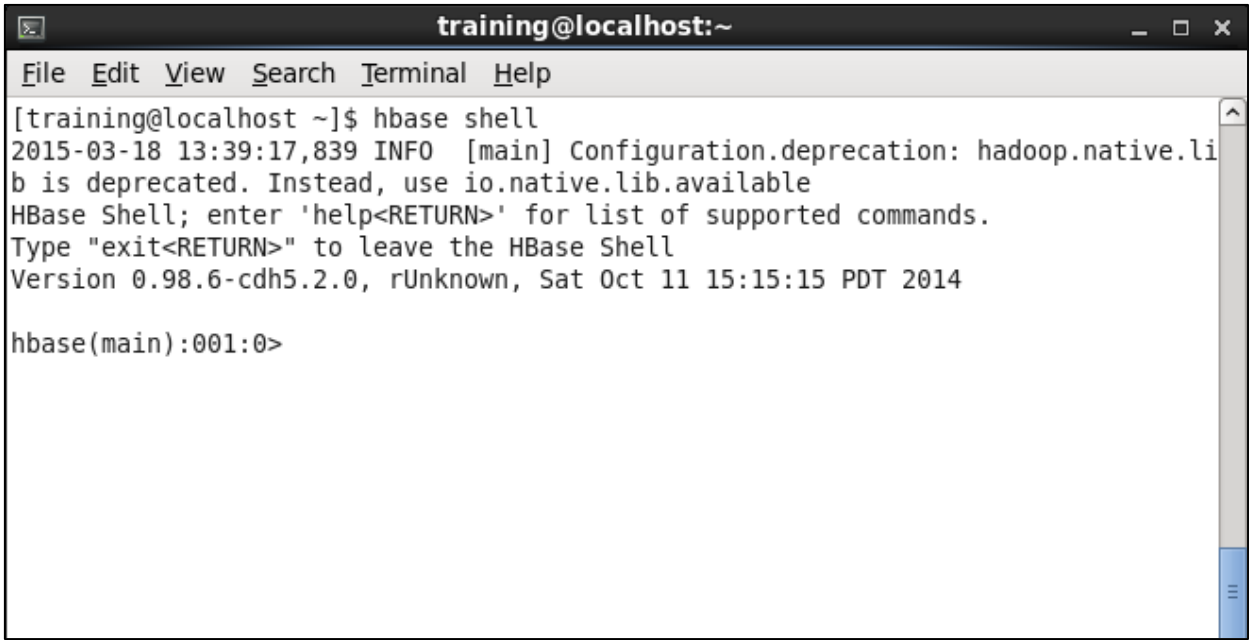
- **Steps for designing an RDBMS schema**
  - Determine all the types of data to be stored
  - Relationship-centric: Determine relationships between data elements
  - Create tables, columns, and foreign keys to maintain relationships
- **Steps for designing an HBase schema**
  - Data-centric: Identify ways in which data will be accessed
  - Identify types of data to be stored
  - Create data layouts and keys

HBase Shell Basics

- **The HBase Shell is an interac=ve shell for sending commands to HBase**
- **HBase shell uses JRuby**
  - Wraps Java client calls in Ruby
  - Allows Ruby syntax to be used for commands
  - Makes parameter usage a liBle different than most shells
    - Command parameters are single quoted (')

Running the HBase Shell

```
$ hbase shell
```



Basic Commands

- **Get help:**

```
hbase> help
```

- **Get HBase status:**

```
hbase> status
3 servers, 0 dead, 1.3333 average load
```

- **Get version:**

```
hbase> version
0.98.6-cdh5.2.0, rUnknown, Sat Oct 11 15:15:15 PDT 2014
```

Shell Command Syntax

- **Shell commands are oKen followed by parameters**

```
hbase> command 'parameter1', 'parameter2'
```

- **More advanced parameters use Ruby hashes**
  - Ruby hash syntax is **{*PARAM => 'stringvalue'*}**
  - The Ruby '**=>**' operator is called a hash rocket

- **Commands with advanced parameters look like this:**

```
hbase> command 'parameter1', {PARAMETER2 => 'stringvalue',
PARAMETER3 => intvalue}
```

- **Example:**

```
hbase> create 'movie', {NAME => 'desc', VERSIONS => 5}
```

Shell Scripting

- **The HBase Shell works in interac=ve and batch modes**
  - Allows a script to be wriBen in JRuby/Ruby and passed into the shell
- **Passing scripts to the HBase Shell**

```
$ hbase shell pathtorubyscript.rb
```

- **Shell scripts can take command line parameters to expand func=onality**

```
hbase> require 'pathtorubyscript.rb'

hbase> myRubyFunction 'parameter1'
```

HBase Table Creation

- **Only tables and column families need to be specified at table crea=on**
  - You must supply names for the table and column family/families
  - Every table must have at least one column family
  - Any optional se^ n gs can be changed later
- **Through a new namespace feature, HBase tables can be grouped**
  - Similar to the “database” or “schema” concept in relational database systems
- **Table crea=on is different from tradi=onal RDBMS table crea=on**
  - No columns or strict relationships need to be created
  - No constraints or foreign key relationships are specified
  - No database namespace needs to be supplied

Creating Tables

- **General form:**

```
create 'tablename', {NAME => 'colfam' [, options]} [,{...}]
```

- **Examples:**

```
create 'movie', {NAME => 'desc'}

create 'movie', {NAME => 'desc', VERSIONS => 2}
create 'movie', {NAME => 'desc'}, {NAME => 'media'}
```

- **Shorthand (with default op=ons):**

```
create 'movie', 'desc', 'media'
```

Managing HBase Namespaces

- **HBase provides the capability to define and manage namespaces**
- **Mul=ple tables can belong to a single namespace**
  - Define the table's namespace when the table is created
- **Namespaces can be created, dropped, or altered:**

```
create_namespace 'namespaceName'

drop_namespace 'namespaceName'

alter_namespace 'namespaceName', {METHOD => 'set',
'PROPERTY_NAME' => 'PROPERTY_VALUE'}
```

- **There are two predefined special namespaces**
  - **`hbase`** – A system namespace that holds HBase internal tables
  - **`default`** – A namespace for tables with no explicit namespace defined

Creating Tables in Namespaces

- **General form:**

```
create 'namespace:tablename', {NAME => 'colfam' [, options]}
    [,{...}]
```

- **Examples:**

```
create_namespace 'entertainment'

create 'entertainment:movie', {NAME => 'desc'}
```

Listing and Describing Tables

- **List all the tables in HBase**

```
hbase> list
```

- **Give extended details about a table**
  - Provides all column families in a table, their properties, and values

```
hbase> describe 'movie'
```

Disabling and Enabling Tables

- **Disabling a table puts it in a maintenance state**
  - Allows various maintenance commands to be run
  - Prevents all client access
  - May take up to several minutes for a table to disable

```
hbase> disable 'movie'
```

- **Take the table out of maintenance state**

```
hbase> enable 'movie'
```

Deleting

- **The `drop` command removes the table from HBase and deletes all its files in HDFS**
  - The table must be disabled first

```
hbase> disable 'movie'

hbase> drop 'movie'
```

- **The `truncate` command deletes every row in the table**
  - Table and column family schema are unaffected
  - It is not necessary to manually disable the table first

```
hbase> truncate 'movie'
```

Altering Tables

- Tables can be changed after creation
  - The entire table can be modified
  - Column families can be added, modified, or deleted
- For some operations, tables must be disabled while the changes are being applied
  - Re-enable the table after changes are complete
  - As of HBase 0.92, schema changes such as column family changes do not require the table to be disabled
  - By default, online schema changes are enabled
    - The `hbase.online.schema.update.enable` property is set to `true`

Adding, Deleting, and Modifying Column Families

- Add a column family to an existing table

```
hbase> alter 'movie', NAME => 'media'
```

-

```
hbase> alter 'movie', NAME => 'media', METHOD => 'delete'
```

-

```
hbase> alter 'movie', NAME => 'desc', VERSIONS => 5
```

Altering Column Families Asynchronously

- Use the shell command `alter_async` to alter a column family
  -
  -

```
hbase> alter_async 'movie', NAME => 'desc', VERSIONS => 6
```

-