PART 0. INTRODUCTION
2. A PARADIGM FOR BIG DATA

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

This material is built based on

Typical problems for scaling traditional databases
- Suppose that the application should track the number of page views for any URL a customer wishes to track
  - The customer’s web page pings the application’s web server with its URL every time a pageview is received
  - Application tells you top 100 URLs by number of pageviews

<table>
<thead>
<tr>
<th>ID (integer)</th>
<th>User_id (integer)</th>
<th>url (varchar(255))</th>
<th>Pageviews(bigint)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FAQs
- Wait list
- Term project topics

Scaling with a queue
- Direct access from Web server to the backend DB cannot handle the large amount of frequent write requests
- Timeout errors
- Batch many increments in a single request

Lambda Architecture
Scaling by sharding the database

- What if your data amount increases even more?
  - Your worker cannot keep up with the writes
- What if you add more workers?
  - Again, the database will be overloaded

- Horizontal partitioning or sharding of database
  - Uses multiple database servers and spreads the table across all the servers
  - Chooses the shard for each key by taking the hash of the key modded by the number of shards
- What if your current number of shards cannot handle your data?
  - Your mapping script should cope with new set of shards
  - Application and data should be re-organized

Other issues

- Fault-tolerance issues
  - What if one of the database machines is down?
  - A portion of the data is unavailable
- Corruption issues
  - What if your worker code accidentally generated a bug and stored the wrong number for some of the data portions

How will Big Data techniques help?

- The databases and computation systems used in Big Data applications are aware of their distributed nature
  - Sharding and replications will be considered as a fundamental component in the design of Big Data systems
- Data is dealt as immutable
  - Users will mutate data continuously
    - The raw pageview information is not modified
- Applications will be designed in different ways

Lambda Architecture

- Big Data systems as a series of layers
  - Batch layer
  - Serving layer
  - Speed layer
- Batch view
  - Precomputed query function
  - Quick access to the values you need
  - batch view = function(all data)
  - query = function(batch view)

Generating batch views

- Batch layer is often a high-latency operation
- All Data
- Batch layer
- Batch view
- Batch view
- Batch view

Generating batch views

- e.g. A function on all the pageviews to precompute an index from a key of [url, day] to count the number of pageviews for that URL for that day
- This index can be used to sum up the counts to get the results

Batch layer

- Batch layer
  - Stores the master copy of the dataset
  - Precomputes batch views
    - The component that performs the batch view processing
  - Stores an immutable, constantly growing master dataset
  - Computes arbitrary functions on that dataset
    - Batch-processing systems
      - e.g. Hadoop, Spark, TensorFlow

```java
Api.execute(Api.hfsSeqfile("/tmp/pageview-counts"),
new Subquery("url", "count")
.predicate(Api.hfsSeqfile("/data/pageviews"),
"url", "user", "timestamp")
.predicate(new Count(), "count");
```
Serving layer

- The batch layer emits batch view as the result of its functions
  - These views should be loaded somewhere and queried
- Specialized distributed database that loads in a batch view and makes it possible to do random reads on it
- Batch update and random reads should be supported
  - e.g. BigQuery, ElephantDB, Dynamo, MongoDB

Speed layer

- Is there any data not represented in the batch view?
  - Data came while the precomputation was running
    - With fully real-time data system
  - Speed layer looks only at recent data
    - Whereas the batch layer looks at all the data at once
    - real time view = function(realtime view, new data)

Lambda architecture

How long should the real time view be maintained?

- Once the data arrives at the serving layer, the corresponding results in the real-time views are no longer needed
  - You can discard pieces of the realtime views

Extended example with Lambda architecture

- Web analytics application tracking the number of pageviews over a range of days
  - The speed layer keeps its own separate view of [url, day]
    - Updates its views by incrementing the count in the view whenever it receives new data
  - The batch layer recomputes its views by counting the pageviews
  - To resolve the query, you query both the batch and realtime views
    - With satisfying ranges
    - Sum up the results

What if the algorithm is not incremental?

- Brain Storming Quiz
  - What are the examples of non-incremental algorithms?
  - How can the lambda architecture handle the non-incremental analysis?
What if the algorithm is not incremental?

- The batch/speed layer will split your data
  - The exact algorithm on the batch layer
  - An approximate algorithm on the speed layer

- The batch layer repeatedly overrides the speed layer
  - The approximation gets corrected
  - Eventual accuracy

Example of a non-incremental algorithm

- Cardinality estimation
  - Count-distinct problem: finding number of distinct elements
  - Counting exact unique counts in the batch layer
  - A Hyper-LogLog as an approximation in the speed layer
  - Batch layer corrects what’s computed in the speed layer
  - Eventual accuracy

Recent trends in technology (1/3)

- Physical limits of how fast a single CPU can go
  - Parallelize computation to scale to more data
  - Scale-out solution

- Elastic clouds
  - Infrastructure as a Service (IaaS)
  - Rent hardware on demand rather than owning your hardware
  - Increase and decrease the size of your cluster nearly instantaneously
  - Simplifies system administration

Recent trends in technology (2/3)

- Open source ecosystem for Big Data
  - Batch computation systems
    - Hadoop, HDFS
    - Spark, RDD
  - Serialization frameworks
    - Serializes an object into a byte array from any language
    - Deserializes that byte array into an object in any language
    - Thrift, Protocol Buffers, and Avro

Recent trends in technology (3/3)

- Open source ecosystem for Big Data- cont.
  - Random-access NoSQL databases
    - Sacrifice the full expressiveness of SQL
    - Specializes in certain kinds of operations
    - Cassandra, HBase, MongoDB, etc.

  - Messaging/queuing systems
    - Sends and consumes messages between processes in a fault-tolerant manner
    - Apache Kafka

  - Real-time computation system
    - High throughput, low latency, stream-processing systems
    - Apache Storm

Mapping course components
PART 0. INTRODUCTION
3. DATA MODEL FOR BIG DATA
  : APACHE THRIFT

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

Why we are looking at Thrift

- Applications and services involve multiple, distributed components
  - Possibly developed in different languages
  - Communicate very intensively with each other
  - The wire formats used for data interchange may evolve over time

- Goal
  - Support this interoperability and evolution of wire formats
  - But without compromising on performance (i.e., speed)

Apache Thrift

- A framework for creating interoperable and scalable services
  - Data serialization framework
  - Originally developed at Facebook
  - Now an Apache project

- Users can create their services via a simple Interface definition language (IDL)
  - Consumable and serviceable by numerous languages
  - Codes for clients and servers are automatically generated

- Binary communication protocol
  - Compact size

Thrift Architecture

- You can change the protocol and transport without regenerating codes

Source:

Types

https://thrift.apache.org/docs/types
### Base types
- `bool` - A boolean value, true or false
- `byte` - A signed byte
- `i16` - A 16-bit signed integer
- `i32` - A 32-bit signed integer
- `i64` - A 64-bit signed integer
- `double` - A 64-bit floating point number
- `string` - A text string encoded using UTF-8 encoding

Note the absence of unsigned integer types. This is due to the fact that there are no native unsigned integer types in many programming languages.

### Special Types
- `Binary` - a sequence of unencoded bytes
  - Added to provide better interoperability with Java

### Structs
- Defines a common object to be used across languages
- Equivalent to a class in object oriented programming languages
  - But without inheritance
- Contains a set of strongly typed fields
  - With a unique name identifier within the struct

```
Struct Example {
  1:i32 number=10,
  2:i64 bigNumber,
  3:double decimals,
  4:string name="thrifty"
}
```

### Containers
- Strongly typed data-type that maps to commonly used and commonly available container types in most programming languages.

- `List` - An ordered list of elements
  - e.g. translated to Java `ArrayList`
- `Set` - An unordered set of unique elements
  - e.g. translated to Java `HashSet`, set in Python, etc.
- `List map` - A map of strictly unique keys to values
  - e.g. translated to Java `HashMap`, Python/Ruby dictionary

### Exceptions
- Functionally equivalent to structs
  - Except that they inherit from the native exception base class as appropriate in each target programming language
## Services

- Defined using Thrift types
- Consists of a set of named functions
- With a list of parameters and return types

```thrift
service <name> {
  <returntype><name>(<arguments>)
  [throws (<exceptions>)}
}
```

```thrift
service StringCache{
  void set(1:i32 key, 2:string value),
  string get(1:i32 key) throws (1:KeyNotFound knf),
  void delete(1:i32 key)
}
```

## Transport

### TTransport Interface (1/2)

- Describes "how" the data is transmitted
- Thrift decouples the transport layer from the code generation layer
- Needs to know how to read and write data
  - The origin and destination of the data are irrelevant
  - It may be a socket, a segment of shared memory, or a file on the local disk

```java
TTransport transport = new TTransport();
```

### TTransport Interface (2/2)

- `open` opens the transport
- `close` closes the transport
- `isOpen` indicates whether the transport is open
- `read` reads from the transport
- `write` writes to the transport
- `flush` forces any pending writes

### TServerTransport interface

- `TServerTransport` interface accepts or creates primitive transport objects

- `open` opens the transport
- `listen` begins listening for connections
- `accept` returns a new client transport
- `close` closes the transport

### TSocket

- Implementation of `TServerTransport` interface
- Provides a common, simple interface for a TCP/IP stream socket
- Implemented across all target languages
TFileTransport

- Abstraction of an on-disk file to a data stream
- It can be used to write out a set of incoming Thrift requests to a file on disk