Motivating Examples

Big Data Processing

Recall our definition of "big data":

> Big Data is *any data collection* that is larger than the storage capacity of a single computer system on which it is to be processed.

Notes:

- This is a relativistic definition. What is "big data" in one setting, will not be "big data" in a different setting.
- This is a very permissive definition. A lot of data collections become "big data" under this definition.
- This definition postulates that Big Data is not new - it existed always.

Having said this, we concentrate on modern Big Data processing tasks. We define "modern" as *"have emerged in the past 15-20 years"* (roughly since the advent of the world wide web).

There is a reason for defining "modern" this way. A lot of big data we consider originated via the world wide web.

Specifying Examples

For each example, we specify three things:

1. **Data:** what data collection is used for processing.
2. **Questions:** what do the end users want to find out about the data.

3. **Challenge:** why achieving this is difficult and requires distributed computing for success.

Note: the list of examples below is **by far not exhaustive**.

Most of the examples come from the workflow or data flow associated with a core service provided by the world's largest companies with web services.

Example 1: The "Facebook" Example

Overview: With over 1 billion user accounts, Facebook is one of largest Internet properties in the world. In this example we concentrate of Facebook's core service: display of the current account information ("Facebook wall") of a given user. This service needs to be delivered when

- a user logs onto their Facebook account
- a user clicks on a username of another user on one of Facebook's pages

At each moment of time, it is safe to assume that tens of millions of users are viewing their Facebook pages. The service to deliver the page must have very low latency, and must scale to millions of simulteneous requests.

Data: Facebook user account data is a complex record that identifies a variety of attributes associated with a user account: the full name of the user, their location, status, preferred content settings; as well as contains lists of their Friends, Followers, likes and wall posts (at the very least). This information can be represented as a large complex object keyed to the username on the account, or, to the *unique internal id* associated with the account. The data collection is the over 1 billion of specially prepared records keyed to unique user Id.

Task(s): The core task Facebook has to complete is specified as follows: *"Given a Facebook username, retrieve the user record and render it in HTML."* The latter part of the task ("render it in HTML") is performed by the Facebook web application, that knows how to format and display every single piece of information contained in the user account object.

In this example, we concentrate on the first part of the task:

> *Given a Facebook username, retrieve the appropriate user record.*

Challenge: Why is this task difficult? It appears a simple unique key search, and there are a lot of ways to implement it efficiently. For example, storing all keys together with pointers to the location of the user record on disk in a hash table is a well-known approach.

Facebook has over 1 billion user account records and on the order of tens of millions (per our guess) active users at any given time. The records cannot be stored on a single computer system and they cannot be retrieved efficiently from a single computer system.

In fact, even the hash table won't fit the capacity of a single computer system (and if it did, this solution would still not achieve the correct throughput).

What to do: Store data in a distributed data store across multiple computer systems. The distributed data store must solve well the problem of finding a record by key.

Example 2: The "Google" Example

Overview Google's core business is web search. To support it, Google indexes the billions of web pages (identified by their URL). When an end user enters a search term in Google's search field and clicks "Enter", Google has less than 1 second to retrieve the first 10 results (and to figure out roughly how many total results the search will yield).

Data: Google has an index of web pages - with the content of each web page keyed to the URL of the page. Additionally, Google has an inverted index of the web pages: for each word deemed an appropriate search term, Google stores the list of URLs it occurs in.

Tasks: There are a few tasks we can identify here.

- **Task 1:** Create an inverted index during the web crawl. Even ignoring the fact that the web crawler itself needs to be run in a distributed way (otherwise it will never finish its work), we can view this problem as follows: a constant stream of $\langle$ URL, HTML text $\rangle$ pairs is being sent to a process that needs to build an inverted index.
- **Task 2:** Information retrieval: given a collection of search terms, generate a list of documents that contain all of them.

Challenge: The challenge, again, is the sheer amount of data. According to [1], in early 2013:

- Google indexed around **30 trillion** individual web pages.
- Google's single index of these pages occupies around 1000 Terabytes (just under a Petabyte) of space.
- Google goes through the process of rebuilding the index multiple times each month.

Analysis: The inverted index construction problem is basically a problem of data transformation. The keyword search problem is a combination of search (see the "Facebook" example) and list merge tasks.

Example 3: The "Twitter" Example

Overview: Twitter's core service is showing short messages written by people to their followers and to the world. According to [2], Twitter handles on the order of hundreds of millions of accounts, with a total of 300 billion tweets over the lifetime of the service, and with current velocity of about 500 million individual tweets per day.

Data: Twitter has two core data collections: a list of users (over 550 million accounts[2]), and the list of tweets, keyed to the userid as well as to the time stamp.

Task: Given a user name, put together a list of most recent tweets to push to a Twitter delivery platform (web, mobile, etc.)

Challenge: This is a selection/filtering operation that also requires a sort by the publication time. Storing data in sorted order by publication time is not a difficult thing to do (it is basically a list with a *push* insertion method), *however* if all or most retrieval commands demand the most recent data, storing all most recent data on a single system is a **bad** idea.

Analysis: The core challenge internally is to make sure that the workload is balanced. This means that a better organization is to store tweets by user name (i.e., all tweets of the same individual are stored in the same disk space) in reverse chronological order. This reduces the problem to that from the "Facebook" example - find user account among the millions of account quickly and collect the most recent tweet for it.

Example 4: The "Census" Example.

Overview: This example actually comes from the 1980s, when Herman Hollerith built an electric punch-card tabulating machine for the US Census.

Data: Records about individuals. Each census record contains information about one person: name, city, county, state the live in, their gender, date of birth, race and and national origin, creed, and so on - answers to all the census questions.

Task: Tabulate the data in a way that allows to obtain fast counts of people in each category (men/women, people born in each year, people living in each city and state, and so on).

Analysis: In 1890, the census collected information about over 60 million individuals. Without a ready data storage, tabulating all this data by hand would have taken tens of years. Instead, Herman Hollerith built a punch card recording/reading/tabulating system that collected information and automatically categorized it. It took about two years to punch data about 60+ million people onto the punch cards, but all tabulation was performed automatically by running stacks of punch cards through the tabulating machines and adding up the numbers.

Challenge: Assuming that in modern times all records are already stored electornically, we are still looking at a collection of hundreds of millions of records over which a variety of grouping (break the records into groups based on values of one or more attributes) and aggregation operations (count, sum, average) need to be run.

Because the data is stored over a large number of computer systems, a distributed framework solving this problem needs to put together an efficient procedure for both grouping the data (note that with multiple ways in which the data can be grouped, it is impossible to store the data in those groups without some massive data duplication) and aggregation over multiple compute nodes.

Example 5: The "Bioinformatics" Example

Overview: Modern DNA sequencing methods (otherwise known as "Next-Generation Sequencing" or "NGS") work roughly as follows: a "cocktail" of DNA molecule fragments is analyzed, and around 500,000 relatively short DNA fragments are returned on each sequencing run.

Data: A large collection (millions, tens of millions, hundreds of millions) of DNA fragments. Each fragment is a short (about 2Kb) string in an $\{A, T, C, G\}$ alphabet.

Task: Merge DNA fragments into longer DNA sequences. To do so, for each pair of DNA sequences an *alignment* must be performed to determine if an end of one sequence matches an end of another sequence.

Challenge: With about 1 million DNA fragments, comparing every pair to each other creates 1 trillion pairwise comparisons. So, even if the data can be put under control (1 million DNA fragments can be stored on a single disk), the computation cannot.

Analysis: This is an example of a *join* problem - given a data collection (or two separate collections), pair up the data items in the collection based on some *join condition*. In the case of DNA sequence assembly, the join condition is something like "10 or more characters on the end of one sequence match exactly 10 or more characters at the beginning of the other sequence".

Common Themes

The examples above share some features.

1. The task is relatively simple in nature and is easy to perform on small datasets.
2. The task is expected to finish in reasonable time. In some cases "reasonable time" is defined as "under 1 second", in other cases it is on the order of minutes or hours, but in all cases, a single system performing the task will fail to do it on time.
3. In many tasks the data may be complex (a large record), but very few of its fields get used (e.g., only the Id is needed to complete the task, everything else can be viewed as a black box).

References

[1] John Koetsier, (2013) How Google searches 30 trillion web pages, 100 billion times a month, *Venturebeat blog*, March 1, 2013, http://venturebeat.com/2013/03/01/how-google-searches-30-trillion-web-pages-100-billion-times-a-month/.

[2] Craig Smith, (2015), By The Numbers: 170+ Amazing Twitter Statistics, *DMR blog*, http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/, December 15, 2015.