Mining Big Data

Mining Data Streams
(Chapter 4)

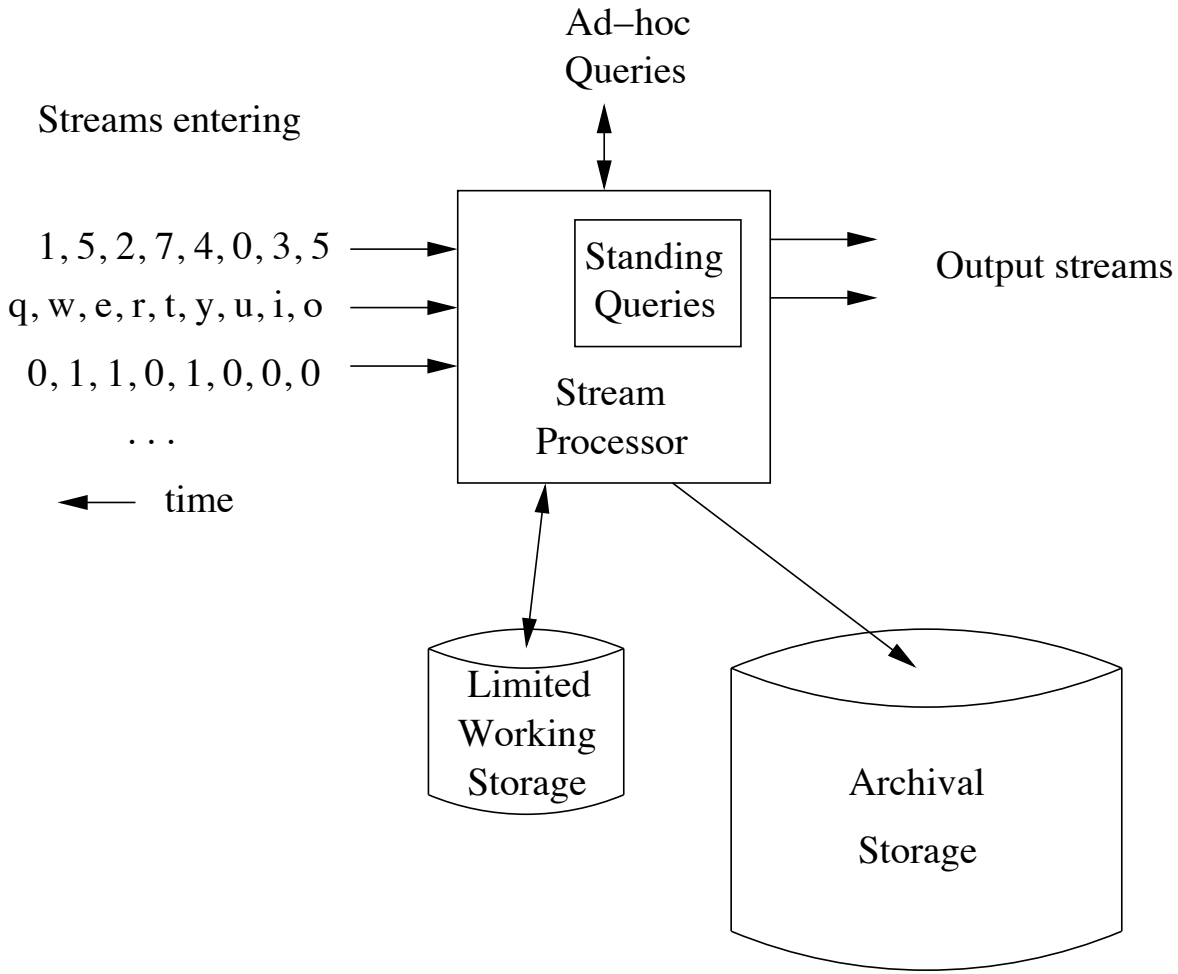
Introduction

- We want to process data that arrives over time at a high frequency.
- If it is not processed immediately or stored then the data is lost forever.
- We assume that data arrives so rapidly that it’s not possible to store it all in the active storage.
- We have to summarize the data seen in some way.

Stream Data Model

- So far, we considered data stored in a database.
- Now we consider streams of data and how to process them.
- We can view a stream processor as a kind of data-management system.
- Any number of streams can enter the system.
- Each stream can provide elements at its own schedule.

Data-Stream Management System



2 types of queries:

1. Standing queries
2. Ad-hoc queries

Figure 4.1 in Rajaramam/Ullman

Stream Data Model

- Arrival rate of stream elements is not under control of the system.
- Streams may be archived in an archival store, but we assume that it's not possible to answer queries from the archival store.
- The archival store can only be examined under special circumstances using time-consuming retrieval processes.
- Working store (disk or main memory) can be used to answer queries and may contain summaries or parts of streams.
- Working store is of limited capacity and can not store all the data from all streams.

Examples of Stream Sources

Stream data arises in many occasions naturally.

Sensor data:

- GPS sensor in the ocean reporting temperature (low data rate and easy to handle).
- GPS sensor reporting surface height in the ocean.
- Surface height might change rapidly and sensor might send back data every tenth of a second.
- We might want to deploy a million sensors to monitor the oceans which results in 3.5 terabytes of data every day.

Examples

Image data:

- Satellites often send down to earth streams of many terrabytes of images per day.
- Surveillance cameras produce streams of images. London is said to have six million of such cameras each producing a stream.

Examples

Internet and Web Traffic:

- Websites receive streams of various types.
- Google receives several hundred millions of search queries each day.
- Yahoo! accepts billions of “clicks” per day on its various sites.
- Many interesting things can be learned from these streams.
- For example: increase in queries like “sore throat” can enable us to track spread of a virus.

Stream Queries

- There are two ways that queries get asked about streams.
- Standing queries (see Fig 4.1) are permanently executed and produce outputs at appropriate times.
- Ad-hoc queries ask a question about the current state of streams.

Examples: Standing Queries

- The stream produced by the ocean-surface-temperature sensor might have a standing query to output an alert whenever the temperature exceeds 25 degrees.
- Standing query might produce the average of the 24 most recent readings whenever a new reading arrives.
- Query might ask for maximum temperature ever recorded.

Ad-hoc Queries

- We can not answer arbitrary queries about streams if we can't store all stream entirely.
- We can prepare for queries by storing appropriate parts or summaries of streams.
- A common approach is to store a sliding window for each stream in the working store.
- Sliding window can be the n most recent elements or all elements arrived within the last t time units.

Example: Ad-hoc Queries

- Web sites often report the number of unique users over the past month.
- If we think of each login as a stream element, we can maintian a window of all logins in the most recent month.

Issues in Stream Processing

- Streams deliver elements very rapidly.
- We must process elements in real time otherwise we lose the ability to process them at all (without accessing archival storage).
- Implies that often the stream-processing algorithm is executed in main memory with no/rare access to secondary storage.
- Many problems on streaming data become hard because of not enough (fast) memory.

Approaches

- It’s often more efficient to get an approximate answer than an exact solution.
- Hashing turns out to be useful and randomness is very helpful to produce good approximate answers.

Sampling Data

- We first want to extract reliable samples from a stream.
- The “trick” is to use hashing in an unusual way.

- We want to have a technique that allows ad-hoc queries on the sample

Example

- A search engine receives a stream of queries and we would like to study the behavior of typical users.
- We assume that the stream consists of tuples (user, query, time).
- We want to answer queries such as "What fraction of the typical user's queries were repeated over the past month?"
- We wish to store only $1/10^{th}$ of the stream elements.

Example: Fraction of repeated user queries

First approach: Store each query with probability 1/10.

- Implies that each user has on average $1/10^{th}$ of its queries stored. (some noise due to randomness)
- Suppose a user has issued s search queries once, d search queries twice, and no search query more than twice.
- Fraction repeated search queries is d/(s+d)

In the sample, we see

- s/10 of the s search queries issued once
- Of the d search queries, we see d/100 twice and
  (2/10) * (9/10)*d=18d/100 once.
  (one of the two is selected (probability of 2/10) and the other is not selected (probability 9/10))
- This gives (d/100)/(d/100 + (s/10+ 18d/100)=d/(10s+19d) as the fraction of repeated search

A Representative Sample

- We can not answer the query by taking a sample of each user's search queries.

Second approach: Pick $1/10^{th}$ of the users at random and take all their search queries. Take no query from a non-selected user.

- We assume that can store a list of all users and flag whether or not they are in the sample.
- Each time a search query arrives, we check whether the users is in the sample and if so we add the search query to the sample (otherwise not).
- If we have no record of having seen this user before, we add the user to the list and flag him with probability 1/10 of being in the sample.

A Representative Sample

- List of users works well if it fits into main memory.
- We can avoid keeping a list of users by using hash functions.
- The hash function maps each user to a bucket 0, …,9. If a user is mapped to bucket 0, he is used for the sample (otherwise not).
- When a new search query arrives we map the user based on the hash function and include the query in the sample if the user is mapped to bucket 0.
- Hash function is used as a random number generator with the important property that if it is applied to the same user several times, we always get the same result.
- More general: we can obtain a sample consisting of any rational fraction a/b of the users by hashing user name to b buckets 0, …, b-1 and adding the search query to the sample if the hash value is less than a.

General Sampling Problem

- Assume our stream consists of tuples with n components.
- A subset of the components are the key component on which the selection of of the sample is based.
- In our example we had three components (user, query, time) and user as key component.
- To take a sample of size a/b, we hash the key value for each tuple to b buckets and take the tuple for the sample if the hash value is less than a.
- If the key consists of more than one component, the hash function needs to combine the components to make a single hash value.
- The selected key values will be a sample of approximately a/b of all the key values appearing in the stream.

Varying Sample Size

- The sample will grow as more of the stream enters.
- If we got a fixed budget on how many tuples can be stored, the fraction of keys must vary (lowering with time).
- We can choose a hash function from the keys to a very large number of values 0, …, B-1 and maintain a threshold t at each time.
- At each time the sample consists of all tuples with keys K for which $h(K) \leq t$ holds.
- If sample size becomes too big, we can lower t to t-1 and exclude all samples with key K and $h(K)=t$.
- We can also lower t by more than 1 and and remove samples with several of the highest hash values.

Filtering Streams

- Another common process on streams is selection/filtering
- We want to accept tuples in the stream that meet a criterion.
- Accepted tuples are passed to another process as a stream and the other tuples are dropped.
- If the selection criterion can directly be calculated on the tuple (e.g. first component less than 10), this is easy.
- Problem becomes harder when the criterion involves look-up for membership in a set (especially if set doesn't fit into main memory)

Example 4.3: Bloom Filtering

- Suppose we have a set S of one billion allowed email addresses that we believe not to be spam.
- Stream consists of pairs (email address, content)
- Suppose, we have one gigabyte of main memory (room for eight billion bits).

Bloom Filtering:

- We use main memory as a bit array and devise a hash function h from email addresses to eight billion buckets.
- Hash member S to a bit and set that bit to 1. All other bits remain 0 ($1/8^{th}$ of bits will be 1).
- When an email arrives we hash the email address and if it hashes to 1, we let the email through.
- If email addresses hashes to 0, we drop the element.

Example 4.3: Bloom Filtering

- Approximately $1/8^{th}$ of the stream element whose email addresses are not in S will be hashed to a 1-bit and let through.
- Still significant to eliminate $7/8^{th}$ of spam (total amount of spam is around 80%)
- If we want to eliminate every spam, we need only to check for membership in S those emails that get through the filter (requires secondary storage)

Bloom Filter

A Bloom Filter consists of:

1. An array of n bits, all initially set to 0.
2. A collection of hash functions $h_1, h_2, \ldots, h_k$ (each mapping key values to n buckets (bits))
3. A set S of m key values.

Purpose is to allow through all elements whose keys are in S, while rejecting most of the others.

Bloom Filter

Usage:

- Initialize bit array (all bits set to 0)
- Take each key value in S and hash it using k hash functions.
- Set each bit to 1 that is $h_i(K)$ for some hash function hi and some key value K in S.
- Test a key K that arrives in the stream by checking all $h_1(K)$, $h_2(K)$, …, $h_k(K)$ and let it through if they are all 1-bits in the bit-array (otherwise reject).

Analysis

- If the key value is in S, it will surely pass through the bloom filter.
- If a key value is not in S, it can still pass the test (however what want to have the probability of such false-positive to be small)
- We calculate the probability of a false-positive in dependence of n (number of bits), m=|S|, and k (number of keys).

Analysis

- Suppose we have x targets and we are throwing y darts each hitting one of the targets with equal probability.
- The probability that a given dart will not hit a given target is (x-1)/x.
- The probability that none of the y darts will hit a given target is

$$\left(\frac{x-1}{x}\right)^y = \left(1 - \frac{1}{x}\right)^{x \cdot (y/x)} \approx e^{-y/x}$$

- Viewing each bit as a target and each member of S as a dart, the probability that a bit is 1 is the probability that a target will be hit by at least 1 dart.

Example (Bloom Filtering):

We have m=10^9 elements in S and n= 8*10^9 bits and the probability that a given bit is not set to 1 is $e^{-1/8}$ and the probability that it is set to 1 is $1-e^{-1/8} \approx 0.1175$. (roughly 1/8 as suggested))

Analysis

General situation:

- S has m members, array n bits, and there are k hash functions.
- Number of targets is n and number of darts is y=km.
- Probability that a bit remains 0 is $e^{-km/n}$.
- We want to have the fraction of 0 bits fairly large.
- We might choose k=n/m or less which implies that the probability of a 0 is at least 1/e.
- In general: the probability of a false positive is the probability of a 1-bit raised to the power of k, i. e. $(1-e^{-km/n})^k$.

Example 4.4

- Consider Example 4.3 with 2 hash functions (instead of 1).
- The probability that a bit remains 0 is $e^{-1/4}$.
- The probability to have a false positive is

  $(1-e^{-1/4})^2 \approx 0.0493$ (compared to 0.1175 for one hash function)

Counting Distinct Elements

- Assume stream elements are from some universal set.
- We would like to know how many different elements have appeared in the stream.

Examples for Applications:

- How many unique users has a website (with logins) had in a given month. The universal set is the set of logins and a stream element is generated if someone logs in.
- Google doesn't require log-in, but we might identify a user by the URL from which they send the query. The universal set of URLs could be all logical host names (essentially infinite, but technically number of URLs is limited by 4 billion).

Counting Distinct Elements

- If number of elements fits in main memory, we can keep a list of all elements seen so far (with an efficient search structure such as hash tables or search trees we can quickly add new elements and check whether elements are already present)
- If there are too many elements or too many streams to be processed in parallel, we can't store everything in the main memory.
- We could use more machines and each machine handling only one or several streams.
- We could use secondary memory and try to batch operations on disk blocks.
- We could estimate the number of distinct elements using randomness.

The Flajolet-Martin Algorithm

- We estimate the number of distinct elements by hashing the elements of the universal set to a bit-string that is sufficiently long.
- Length of the bit-string must be sufficient that there are more possible results of the hash function than elements in the universal set (e.g. 64 bits are enough to hash URLs).
- We shall pick many different hash functions and hash each element of the stream using these functions.
- Important property: A hash function applied to the same element always produces the same result (have already seen this).

The Flajolet-Martin Algorithm

Idea:

- The more different elements we see in the stream, the more different hash-values we shall see.
- If we see many different hash-values, it becomes more likely that one of them is unusual.
- In our case “unusual” means that the bitstring ends with many 0’s.
- Whenever we apply a hash function h to an element a, the bitstring h(a) will end with some number of 0s (called tail length)
- Let R be the maximum tail length seen. We use 2^R as an estimate for the number of distinct elements.

The Flajolet-Martin Algorithm

Analysis:

- The probability that a given element a has h(a) ending in at least r 0's is 2^{-r}.
- Suppose that there m distinct elements in the stream.
- The probability that none of them has a tail length of at least r is $(1-2^{-r})^m$.
- We can rewrite this as $\left((1-2^{-r})^{2^r}\right)^{m2^{-r}} \approx e^{-m2^{-r}}$ (r large)

Implications:

- If m is much larger than 2^r, the probability for a tail of length at least r approaches 1.
- If m is much smaller than 2^r, the probability for a tail of length at least r approaches 0.