CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 33: Hash Table Use Cases

Announcements

- PA3 Implementation due Sunday
- Implementation AutoLab coming soon

HashTables as Sets

We've now seen HashTable's as an implementation of Sets

- **HashSet** in Java -> Expected O(1) runtime for **add**, **contains**, **remove**

What about **HashMap**? What is a map??

HashTables as Sets
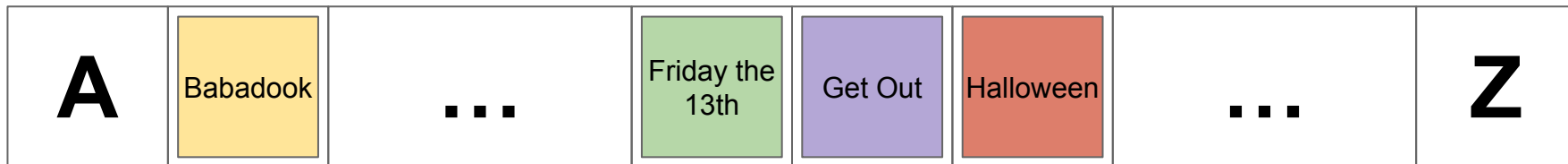
We've now seen HashTable's as an implementation of Sets

- **HashSet** in Java -> Expected O(1) runtime for **add**, **contains**, **remove**

What about **HashMap**? What is a map??

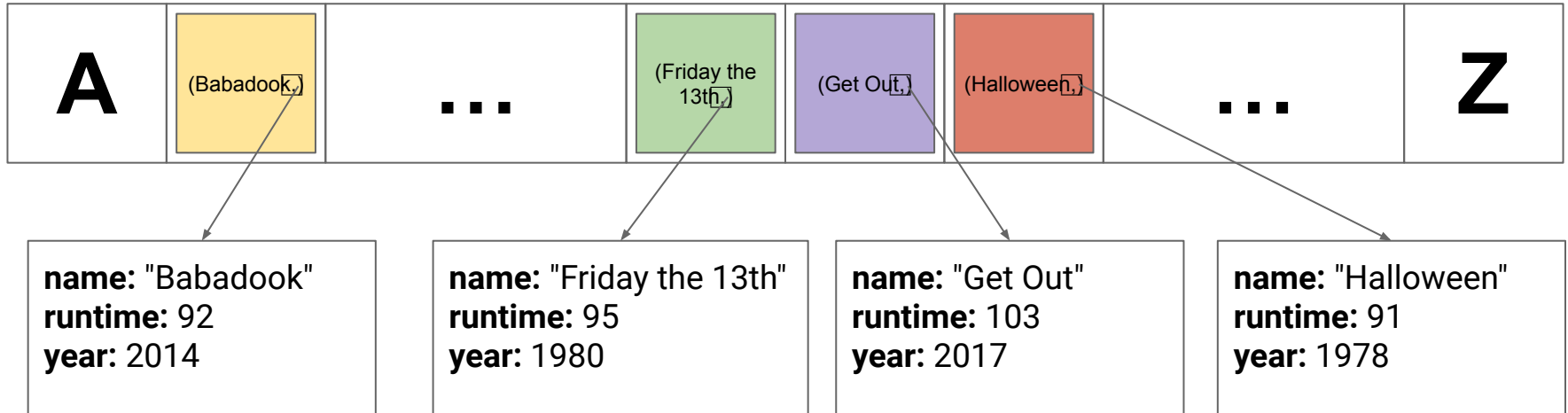- A map IS as set. It is a set of key-value pairs!

HashSets vs HashMaps

This was an example of a `HashSet` that stored movie titles (with a bad hash function…but ignore that for now)

A	Babadook	...	Friday the 13th	Get Out	Halloween	...	Z

HashSets vs HashMaps

This is an example of a `HashMap` that stores key value pairs where the key is a movie title and the value is the movie object associated with that title

| A | (Babadook,) | ... | (Friday the 13th,) | (Get Out,) | (Halloween,) | ... | Z |

name: "Babadook"
runtime: 92
year: 2014

name: "Friday the 13th"
runtime: 95
year: 1980

name: "Get Out"
runtime: 103
year: 2017

name: "Halloween"
runtime: 91
year: 1978

Data Science is Everywhere

- The Corporate World (ie MANGA)
- Open Data → Civic Computing
- Science
- Internet of Things
- …etc

Data is BIG

Remember: $O(f(n))$ tells us the behavior of an algorithm as n gets big

Real world problems are BIG → 100s of MBs, GBs, TBs or more of data
- Think about how much data Facebook, Google, etc have access to
- Recall the OpenData map of Buffalo…that was JUST Buffalo
- How many atoms in a bucket of water? How many stars in the galaxy?
- How many smart devices in this room? at UB?

Data is BIG

Remember: $O(f(n))$ tells us the behavior of an algorithm as **n** gets big

Real world problems are BIG → **100s of MBs, GBs, TBs or more of data**
- Think about how much data Facebook, Google, etc have access to
- Recall the OpenData map of Buffalo…that was JUST Buffalo
- How many atoms in a bucket of water? How many stars in the galaxy?
- How many smart devices in this room? at UB?

Today we'll look at a few common patterns that deal with big data   (that will be especially useful for PA3…)

Usage Pattern 1 (in MANGA)

Dataset: Sales - A sequence of purchase records
- **productID:** Int
- **date:** Date
- **volume:** Int

Objective: Find the 100 most purchased products from the last month

Usage Pattern 1 (in Open Data)

Dataset: Traffic Violations - A sequence of infraction records
- **blockID:** `Int`
- **infraction:** `InfractionType`
- **date:** `Date`

Objective: Find the fraction of parking tickets that were issued in each block over the last year

Usage Pattern 1 (in Science)

Dataset: Vaccinations - Records on COVID vaccination data
- **patientID:** Int
- **doseVolume:** Double
- **contractedCOVID:** Boolean

Objective: Find the dosage that minimizes the rate of contracting COVID

Usage Pattern 1 (in IoT)

Dataset: Train Logistics - Logs related to train travel distances
- **engineID:** Int
- **date:** Date
- **kmTraveledToday:** Double

Objective: If a train engine must be serviced every 30,000km, determine which train engines currently need service

Usage Pattern 1

What do all these use cases have in common?

What basic task do we need to do to meet these objectives?

Usage Pattern 1 - Aggregation!

What do all these use cases have in common?

What basic task do we need to do to meet these objectives?

We need to aggregate data spread across multiple records with a common ID

Usage Pattern 1: Aggregation

Examples:
- "sum up __, for each __"
- "find the average __, by __"
- "count the number of __, for __"
- "what is the biggest/smallest __, for each __"

Pattern:
1. (Optionally) Group records by a common "Group By" key
2. For each group, compute a statistic (ie sum, count, avg, min, max)

Usage Pattern 1: Aggregation

How might we accomplish this efficiently? How much time is required?

Usage Pattern 1: Aggregation

```java
Map<Integer, Integer> groupBySum(List<SaleRecord> records) {
  HashMap<Integer, Integer> result = new HashMap<>();
  for (SaleRecord record : records) {
    result.put(record.productId,
        result.getOrDefault(record.productId, 0) + record.quantity);
  }
  return result;
}
```

An example of the aggregation pattern for the MANGA use case described previously

Usage Pattern 1: Aggregation

```java
1  Map<Integer, Integer> groupBySum(List<SaleRecord> records) {
2    HashMap<Integer, Integer> result = new HashMap<>();
3    for (SaleRecord record : records) {          For each record in the data set…
4      result.put(record.productId,
5          result.getOrDefault(record.productId, 0) + record.quantity);
6    }
7    return result;
8  }
```

An example of the aggregation pattern for the MANGA use case described previously

Usage Pattern 1: Aggregation

```
1  Map<Integer, Integer> groupBySum(List<SaleRecord> records) {
2    HashMap<Integer, Integer> result = new HashMap<>();
3    for (SaleRecord record : records) {       For each record in the data set…
4      result.put(record.productId,   …hash it by the desired key…
5          result.getOrDefault(record.productId, 0) + record.quantity);
6    }
7    return result;
8  }
```

An example of the aggregation pattern for the MANGA use case described previously

Usage Pattern 1: Aggregation

```
1  Map<Integer, Integer> groupBySum(List<SaleRecord> records) {
2    HashMap<Integer, Integer> result = new HashMap<>();
3    for (SaleRecord record : records) {
4      result.put(record.productId,
5          result.getOrDefault(record.productId, 0) + record.quantity);
6    }
7    return result;
8  }
```

For each record in the data set…

…hash it by the desired key…

…and update the value based on the desired aggregation operation (ie sum)

An example of the aggregation pattern for the MANGA use case described previously

Usage Pattern 1: Aggregation

```
1  Map<Integer, Integer> groupBySum(List<SaleRecord> records) {
2    HashMap<Integer, Integer> result = new HashMap<>();
3    for (SaleRecord record : records) {
4      result.put(record.productId,
5          result.getOrDefault(record.productId, 0) + record.quantity);
6    }
7    return result;
8  }
```

Complexity?

An example of the aggregation pattern for the MANGA use case described previously

Usage Pattern 1: Aggregation

```
1  Map<Integer, Integer> groupBySum(List<SaleRecord> records) {
2    HashMap<Integer, Integer> result = new HashMap<>();
3    for (SaleRecord record : records) {
4      result.put(record.productId,
5          result.getOrDefault(record.productId, 0) + record.quantity);
6    }
7    return result;
8  }
```

Complexity? expected $O(|data|)$ (each update is expected $O(1)$)

An example of the aggregation pattern for the MANGA use case described previously

Potential Issues

Issue 1: Data is too big to fit in memory
- ie All of Amazon or Google's users

Potential Issues

Issue 1: Data is too big to fit in memory
- ie All of Amazon or Google's users

Idea: Use disk for storage
- **Problem:** Group-by keys are not in any specific order…
- **Idea:** Do an initial $O(n)$ pass to organize the data

Buffered Writer

Consider a BufferedWriter

It has a fixed size, we can add to it, and when it becomes full, it empties itself to disk...

Buffered Writer

Consider a BufferedWriter

It has a fixed size, we can add to it, and when it becomes full, it empties itself to disk...

Buffered Writer

Consider a BufferedWriter

It has a fixed size, we can add to it, and when it becomes full, it empties itself to disk...

Buffered Writer

Consider a BufferedWriter

It has a fixed size, we can add to it, and when it becomes full, it empties itself to disk...

Hash Partitioning

Create multiple buffered writers for specific keys...

hash(key) % N = 0

hash(key) % N = 1

hash(key) % N = N-1

Hash Partitioning

Create multiple buffered writers for specific keys...

hash(key) % N = 0

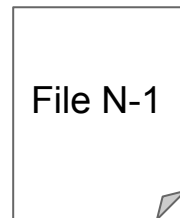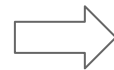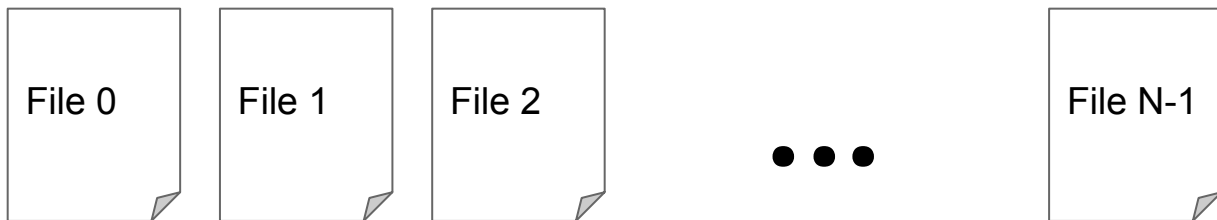hash(key) % N = 1

hash(key) % N = N-1

File N-1

Hash Partitioning

Each writer will result in a file...and all instances of a key will be in the same file

File 0   File 1   File 2   ● ● ●   File N-1

$O(n)$ total writes to disk

Hash Partitioning

Can load a single file and compute aggregate for just that file before moving to the next file

File 0
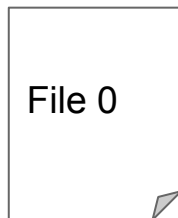
$O(n)$ total reads from disk

Potential Issues

Issue 2: Data is too big to even fit on one computer!

Solution: Use multiple computers (distributed computation)
- **Idea 1:** Compute each aggregate locally, then send those partial results to be aggregated together
- **Idea 2:** Hash partition (shuffle) to each computer then compute locally

Usage Pattern 2 (in MANGA)

Dataset 1: Sales - A sequence of purchase records
- **productID:** Int
- **date:** Date
- **volume:** Int

Dataset 2: Pricing - A sequence of product IDs and their price
- **productID:** Int
- **price:** Double

Objective: Find the 100 products with the highest gross profit

Usage Pattern 2 (in Open Data)

Dataset 1: Traffic Violations - A sequence of infraction records
- **blockID:** `Int`
- **infraction:** `InfractionType`
- **date:** `Date`

Dataset 2: Tax Assessments - A sequence of building tax assessments
- **buildingOwner:** `String`
- **blockID:** `Int`
- **assessment:** `Double`

Objective: Plot total taxes vs number of tickets for a given block

Usage Pattern 2 (in Science)

Dataset 1: Trials - A sequence of vaccination doses
- **patientID:** `Int`
- **doseVolume:** `Double`

Dataset 2: Infections - A sequence COVID infection reports
- **patientID:** `String`
- **date:** `Date`

Objective: Find the dosage that minimizes the rate of contracting COVID

Usage Pattern 2 (in IoT)

Dataset: Train Logistics - Logs related to train travel distances
- **engineID:** `Int`
- **date:** `Date`
- **kmTraveledToday:** `Double`
- **locationID:** `Int`

Dataset 2: Locations - A list of locations with service stations
- **locationID:** `Int`
- **serviceCapacity:** `Int`

Objective: Determine if any locations have more trains in need of service than they have capacity for.

Usage Pattern 2

What do all these use cases have in common?

What basic task do we need to do to meet these objectives?

Usage Pattern 2: Joins

What do all these use cases have in common?

What basic task do we need to do to meet these objectives?

We need to join multiple different datasets to match up corresponding records in each based on some common attribute

Usage Pattern 2: Joins

Examples:
- "combine these datasets"
- "look up __ for each __"
- "join __ and __ on __"

Pattern:
1. For each record in one dataset…
   a. Find the corresponding record(s) in the second dataset
2. Output each pair of matched records

Usage Pattern 2: Joins

How might we accomplish this efficiently? How much time is required?

Nested-Loop Join

```java
List<Record> NLJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
  List<Record> result = new ArrayList<>();
  for (SaleRecord s : sales) {
    for (ProductRecord p : prices) {
      if (s.productId == p.productId) {
        result.add(new Record(s,p));
      }
    }
  }
  return result;
}
```

Nested-Loop Join

```java
List<Record> NLJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
  List<Record> result = new ArrayList<>();
  for (SaleRecord s : sales) {
    for (ProductRecord p : prices) {
      if (s.productId == p.productId) {
        result.add(new Record(s,p));
      }
    }
  }
  return result;
}
```

For each record in the first table…

…search the second table for all records that match on the common key

Nested-Loop Join

```
1  List<Record> NLJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
2    List<Record> result = new ArrayList<>();
3    for (SaleRecord s : sales) {
4      for (ProductRecord p : prices) {
5        if (s.productId == p.productId) {
6          result.add(new Record(s,p));
7        }
8      }
9    }
10   return result;
11 }
```

For each record in the first table…

…search the second table for all records that match on the common key

Complexity?

Nested-Loop Join

```
 1 List<Record> NLJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
 2   List<Record> result = new ArrayList<>();
 3   for (SaleRecord s : sales) {
 4     for (ProductRecord p : prices) {
 5       if (s.productId == p.productId) {
 6         result.add(new Record(s,p));
 7       }
 8     }
 9   }
10   return result;
11 }
```

For each record in the first table…

…search the second table for all records that match on the common key

Complexity? *O*(|sales| * |prices|)

Nested-Loop Join

```
 1  List<Record> NLJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
 2    List<Record> result = new ArrayList<>();
 3    for (SaleRecord s : sales) {
 4      for (ProductRecord p : prices) {
 5        if (s.productId == p.productId) {
 6          result.add(new Record(s,p));
 7        }
 8      }
 9    }
10    return result;
11  }
```

For each record in the first table…

…search the second table for all records that match on the common key

Complexity? *O*(|sales| * |prices|)

Can we do better? What makes this approach so expensive?

Sort Merge Join

Idea: In merge sort, we saw that the combine step only cost $O(n)$ because the two pieces were already sorted…

Sort Merge Join

```
 1  List<Record> SMJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
 2    List<Record> result = new ArrayList<>();
 3    Collections.sort(sales);
 4    Collections.sort(prices);
 5
 6    /* Initialize iterators to the start of both lists */
 7    while (/* More items in both lists */) {
 8      /* If the current ID in each iterator matches, add to result */
 9      /* If not, advance whichever iterator points to the smallest ID */
10    }
11    return result;
12  }
```

Sort Merge Join

```
 1  List<Record> SMJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
 2    List<Record> result = new ArrayList<>();
 3    Collections.sort(sales);
 4    Collections.sort(prices);              Sort both lists by the join key…
 5
 6    /* Initialize iterators to the start of both lists */
 7    while (/* More items in both lists */) {
 8      /* If the current ID in each iterator matches, add to result */
 9      /* If not, advance whichever iterator points to the smallest ID */
10    }
11    return result;
12  }
```

Sort Merge Join

```
 1  List<Record> SMJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
 2    List<Record> result = new ArrayList<>();
 3    Collections.sort(sales);
 4    Collections.sort(prices);
 5
 6    /* Initialize iterators to the start of both lists */
 7    while (/* More items in both lists */) {
 8      /* If the current ID in each iterator matches, add to result */
 9      /* If not, advance whichever iterator points to the smallest ID */
10    }
11    return result;
12  }
```

Perform a variation of merge on both lists to build result

Sort Merge Join

```
 1  List<Record> SMJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
 2    List<Record> result = new ArrayList<>();
 3    Collections.sort(sales);
 4    Collections.sort(prices);
 5
 6    /* Initialize iterators to the start of both lists */
 7    while (/* More items in both lists */) {
 8      /* If the current ID in each iterator matches, add to result */
 9      /* If not, advance whichever iterator points to the smallest ID */
10    }
11    return result;
12  }
```

Perform a variation of merge on both lists to build result

Complexity?

Sort Merge Join

```
 1  List<Record> SMJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
 2    List<Record> result = new ArrayList<>();
 3    Collections.sort(sales);
 4    Collections.sort(prices);
 5
 6    /* Initialize iterators to the start of both lists */
 7    while (/* More items in both lists */) {
 8      /* If the current ID in each iterator matches, add to result */
 9      /* If not, advance whichever iterator points to the smallest ID */
10    }
11    return result;
12  }
```

Perform a variation of merge on both lists to build result

Complexity? *O*(*n* log(*n*))...**but we can still do better**

Hash Join

Final Idea: How can we skip the "search" for common keys?

Hash Join

Final Idea: How can we skip the "search" for common keys? A HashTable!

Hash Join

```java
List<Record> hashJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
  HashMap<Integer, ProductRecord> priceTable = new HashMap<>();
  for (ProductRecord p : prices) {
    priceTable.put(productId, p);
  }
  List<Record> result = new ArrayList<>();
  for (SaleRecord s : sales) {
    if (priceTable.containsKey(s.productId)) {
      result.add(new Record(s, priceTable.get(s.productId));
    }
  }
  return result;
}
```

Hash Join

```java
List<Record> hashJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
  HashMap<Integer, ProductRecord> priceTable = new HashMap<>();
  for (ProductRecord p : prices) {
    priceTable.put(productId, p);
  }
  List<Record> result = new ArrayList<>();
  for (SaleRecord s : sales) {
    if (priceTable.containsKey(s.productId)) {
      result.add(new Record(s, priceTable.get(s.productId));
    }
  }
  return result;
}
```

Build a hash table for the first dataset…

Hash Join

```java
List<Record> hashJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
  HashMap<Integer, ProductRecord> priceTable = new HashMap<>();
  for (ProductRecord p : prices) {
    priceTable.put(productId, p);
  }
  List<Record> result = new ArrayList<>();
  for (SaleRecord s : sales) {
    if (priceTable.containsKey(s.productId)) {
      result.add(new Record(s, priceTable.get(s.productId));
    }
  }
  return result;
}
```

Build a hash table for the first dataset…

…then for each element in the second dataset, **probe** the HashTable to find matches (in expected $O(1)$ time per record)

Hash Join

```
1  List<Record> hashJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
2    HashMap<Integer, ProductRecord> priceTable = new HashMap<>();
3    for (ProductRecord p : prices) {
4      priceTable.put(productId, p);
5    }
6    List<Record> result = new ArrayList<>();
7    for (SaleRecord s : sales) {
8      if (priceTable.containsKey(s.productId)) {
9        result.add(new Record(s, priceTable.get(s.productId));
10     }
11   }
12   return result;
13 }
```

Build a hash table for the first dataset…

…then for each element in the second dataset, **probe** the HashTable to find matches (in expected $O(1)$ time per record)

Complexity?

Hash Join

```java
List<Record> hashJoin(List<SaleRecord> sales, List<ProductRecord> prices) {
  HashMap<Integer, ProductRecord> priceTable = new HashMap<>();
  for (ProductRecord p : prices) {
    priceTable.put(productId, p);
  }
  List<Record> result = new ArrayList<>();
  for (SaleRecord s : sales) {
    if (priceTable.containsKey(s.productId)) {
      result.add(new Record(s, priceTable.get(s.productId));
    }
  }
  return result;
}
```

Build a hash table for the first dataset…

…then for each element in the second dataset, **probe** the HashTable to find matches (in expected *O*(1) time per record)

Complexity? expected *O*(|prices| + |sales|)

Potential Issues

Issue 1: Too much data to fit in memory
- **Solution:** Hash Partition both datasets on the join key

Issue 2: Too much data to fit on one computer
- **Solution 1:** Hash Partition both datasets on the join key
- **Solution 2:** Send only relevant data using a Bloom Filter…

For More Info...

CSE 305: How to build compilers / languages that can easily express common data science patterns

CSE 460: How to organize data to make it easier to find, and apply tricks to make common data science patterns more efficient

CSE 462: How to build systems that automatically pick the best data structure/algorithm for each data science pattern

CSE 486: How to build systems that do these computations at scale