15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 7

Collect, Sets and Tables

Synopsis

- The collect operation
- The map-collect-reduce paradigm
- Sets
- Tables

The Collect Operation

- Group items that share a common key.

$$\texttt{Data} = \langle (\text{“jack sprat”}, \text{“15-210”}), (\text{“jack sprat”}, \text{“15-213”}), (\text{“mary contrary”}, \text{“15-210”}), (\text{“mary contrary”}, \text{“15-251”}), (\text{“mary contrary”}, \text{“15-213”}), (\text{“peter piper”}, \text{“15-150”}), (\text{“peter piper”}, \text{“15-251”}), \ldots \rangle$$

$$\downarrow$$

$$\texttt{rosters} = \langle (\text{“15-150”}, \langle \text{“peter piper”}, \ldots \rangle) (\text{“15-210”}, \langle \text{“jack sprat”}, \text{“mary contrary”}, \ldots \rangle) (\text{“15-213”}, \langle \text{“jack sprat”}, \ldots \rangle) (\text{“15-251”}, \langle \text{“mary contrary”}, \text{“peter piper”} \rangle) \ldots \rangle$$

The Collect Operation

- Very common operation in Relational Databases
- Usually called the Group by operation.

$$\texttt{rosters} = \langle (\text{“15-150”}, \langle \text{“peter piper”}, \ldots \rangle) \\ (\text{“15-210”}, \langle \text{“jack sprat”}, \text{“mary contrary”}, \ldots \rangle) \\ (\text{“15-213”}, \langle \text{“jack sprat”}, \ldots \rangle) \\ (\text{“15-251”}, \langle \text{“mary contrary”}, \text{“peter piper”} \rangle) \\ \ldots \rangle$$

- Students are grouped by Course Numbers.

The Collect Operation

$$\texttt{collect} : (\alpha \times \alpha \to \textit{order}) \to (\alpha \times \beta)\ \textit{seq}$$
$$\to (\alpha \times \beta\ \textit{seq})\ \textit{seq}$$

1. $\alpha \times \alpha \to \textit{order}$ is a function for comparing keys of type α
2. $(\alpha \times \beta)$ *seq* is a sequence of key-value pairs
3. $(\alpha \times \beta\ \textit{seq})\ \textit{seq}$ is the resulting sequence:
   - each unique α value is paired with a sequence of all β values it appears with

The Collect Operation

$$\texttt{collectStrings} = \texttt{collect String.compare}$$

$$\texttt{rosters} = \texttt{collectStrings}(\langle (n, c) : (c, n) \in \texttt{Data} \rangle)$$

$$\begin{aligned}\texttt{rosters} = \langle & (\text{“15-150”}, \langle \text{“peter piper”}, \ldots \rangle) \\ & (\text{“15-210”}, \langle \text{“jack sprat”}, \text{“mary contrary”}, \ldots \rangle) \\ & (\text{“15-213”}, \langle \text{“jack sprat”}, \ldots \rangle) \\ & (\text{“15-251”}, \langle \text{“mary contrary”}, \text{“peter piper”} \rangle) \\ & \ldots \rangle\end{aligned}$$

- $\langle (n, c) : (c, n) \in \texttt{Data} \rangle$ arranges the data appropriately.

The Collect Operation

- How would you implement `collect`?
  - Sort the items on their keys
  - Partition the resulting sequence
  - Pull out pairs between each key change

The Collect Operation

- The dominant cost of `collect` is in sorting.
- Work is $O(W_c n \log n)$, Span is $O(S_c \log^2 n)$
  - W_c work bound for the comparison function
  - S_c span bound for the comparison function
- A $O(n)$ work can be implemented with hashing.
  - Need a separate hash function
  - Output not in sorted order

Using Collect in Map-reduce

- The map-reduce paradigm is used to process very large collection of documents.
  - A document is a collection of words/strings.
  - Not the `mapReduce` of 15-150!
- map-reduce paradigm $\equiv$ map-collect-reduce(s).

Using Collect in Map-reduce

- f_m maps each document to a sequence of key-value pairs.
  - $f_m : (document \rightarrow (key \times \alpha)\ seq)$
- All key-value pairs in a document are collected.
- f_r is applied to the keys to get a single value for a key.
  - $f_r : key \times \alpha\ seq \rightarrow \beta$

An Example

$$\texttt{docs} = \langle \text{“this is a document”}, \text{“this is is another document”}, \text{“a last document”} \rangle$$

$$\downarrow$$

$$\langle (\text{“this”}, 1), (\text{“is”}, 1), (\text{“a”}, 1), (\text{“document”}, 1), (\text{“this”}, 1), (\text{“is”}, 1), (\text{“is”}, 1), (\text{“another”}, 1), (\text{“document”}, 1), (\text{“a”}, 1), (\text{“last”}, 1), (\text{“document”}, 1) \rangle$$

$$\downarrow$$

$$\langle (\text{“a”}, 2), (\text{“another”}, 1), (\text{“document”}, 3), (\text{“is”}, 3), (\text{“last”}, 1), (\text{“this”}, 2) \rangle$$

MapReduce in SML

```
1  fun mapCollectReduce f_m f_r docs =
2      let
3          pairs = flatten⟨ f_m(s) : s ∈ docs ⟩
4          groups = collect String.compare pairs
5      in
6        ⟨ f_r(g) : g ∈ groups ⟩
7      end
```

- $\mathit{flatten}\langle\langle a, b, c\rangle, \langle d, e\rangle\rangle \Rightarrow \langle a, b, c, d, e\rangle$

MapReduce in SML

```
1  fun mapCollectReduce f_m f_r docs =
2      let
3          pairs = flatten ⟨ f_m(s) : s ∈ docs ⟩
4          groups = collect String.compare pairs
5      in
6        ⟨ f_r(g) : g ∈ groups ⟩
7      end
```

```
fun f_m(doc) = ⟨ (w, 1) : tokens doc ⟩
fun f_r(w, s) = (w, reduce + 0 s)
```

MapReduce Example in SML

fun $f_m(\texttt{doc}) = \langle (w, 1) : \texttt{tokens doc} \rangle$

fun $f_r(w, s) = (w, \texttt{reduce} + 0\ s)$

$\texttt{countWords} = \texttt{mapCollectReduce}\ f_m\ f_r$

$\texttt{countWords}\ \langle$"this is a document",
"this is is another document",
"a last document"$\rangle$

$\Rightarrow \langle$("a", 2), ("another", 1), ("document", 3), ("is", 3),
("last", 1), ("this", 2)$\rangle$

Sets

- Sets play a very important role in math.
- Often needed in many algorithms.
- Many languages either support sets directly or have libraries for sets.
- In 15-210 we use a purely functional definition for sets:
  - When updates are done, a new set is returned.

Sets as an ADT

- $\mathbb{U}$ is a universe of elements.
- The SET ADT is a type $\mathbb{S}$ that represents the power set of $\mathbb{U}$.

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{S} & = \emptyset \\
\texttt{size}(S) & : & \mathbb{S} \to \mathbb{Z}_{\geq 0} & = |S| \\
\texttt{singleton}(e) & : & \mathbb{U} \to \mathbb{S} & = \{e\} \\
\texttt{filter}(f, S) & : & ((\mathbb{U} \to \{\texttt{T}, \texttt{F}\}) & = \{s \in S \mid f(s)\} \\
 & & \quad \times \mathbb{S}) \to \mathbb{S} &
\end{array}
$$

Sets as an ADT

$$
\begin{array}{lcll}
\texttt{find}(S, e) & : & \mathbb{S} \times \mathbb{U} \to \{\mathtt{T}, \mathtt{F}\} & = \; |\{s \in S \mid s = e\}| = 1 \\
\texttt{insert}(S, e) & : & \mathbb{S} \times \mathbb{U} \to \mathbb{S} & = \; S \cup \{e\} \\
\texttt{delete}(S, e) & : & \mathbb{S} \times \mathbb{U} \to \mathbb{S} & = \; S \setminus \{e\}
\end{array}
$$

$$
\begin{array}{lcll}
\texttt{intersection}(S_1, S_2) & : & \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = \; S_1 \cap S_2 \\
\texttt{union}(S_1, S_2) & : & \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = \; S_1 \cup S_2 \\
\texttt{difference}(S_1, S_2) & : & \mathbb{S} \times \mathbb{S} \to \mathbb{S} & = \; S_1 \setminus S_2
\end{array}
$$

- What is the relationship between these two groups?

Sets as an ADT

- We do not really need `find`, `insert`, `delete`!

$$
\begin{aligned}
\texttt{find}(S, e) &= \texttt{size}(\texttt{intersection}(S, \texttt{singleton}(e))) = 1 \\
\texttt{insert}(S, e) &= \texttt{union}(S, \texttt{singleton}(e)) \\
\texttt{delete}(S, e) &= \texttt{difference}(S, \texttt{singleton}(e))
\end{aligned}
$$

- `intersection`, `union`, and `difference`
  - can operate on multiple elements, and
  - are suitable for parallelism

Cost Model for Sets

- Underlying data structure can be
  - hash-tables
  - balanced trees
- We will assume a balanced-tree implementation.
- We will assume comparison of two set elements take
  - W_c work and S_c span.

Cost Model for Sets

	Work	*Span*
`size`(S) `singleton`(e)	$O(1)$	$O(1)$
`filter`(f, S)	$O\left(\sum_{e \in S} W(f(e))\right)$	$O\left(\log \lvert S \rvert + \max_{e \in S} S(f(e))\right)$
`find`(S, e) `insert`(S, e) `delete`(S, e)	$O(W_c \cdot \log \lvert S \rvert)$	$O(S_c \cdot \log \lvert S \rvert)$

Cost Models for Sets

$$\texttt{intersection}(S_1, S_2) \quad \texttt{union}(S_1, S_2) \quad \texttt{difference}(S_1, S_2) \Rightarrow \quad Work = O\left(W_c \cdot m \cdot \log(1 + \tfrac{n}{m})\right) \quad Span = O\left(S_c \cdot \log(n + m)\right)$$

$$n = \max(|S_1|, |S_2|) \qquad m = \min(|S_1|, |S_2|)$$

- Sets are equal size ($n = m$)
  - Work $= O(W_c \cdot m \cdot \log(1 + 1)) = O(W_c \cdot n)$
  - Span $= O(S_c \cdot \log n)$
- One of the sets is a singleton ($m = 1$)
  - Work $= O(W_c \cdot \log(1 + n)) = O(W_c \cdot \log n)$
  - Span $= O(S_c \cdot \log(n + 1)) = O(S_c \cdot \log n)$

Tables

- Table is an ADT for sets of key-value pairs.

- $\{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots, (k_n \mapsto v_n)\}$
- $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$

- Each key appears only once
- Many languages provide either built-in support or libraries.

Tables

- $\mathbb{K}$ is the universe of keys.
- $\mathbb{V}$ is the universe of values.
- $\mathbb{T}$ is a type that represents the power set of $\mathbb{K} \times \mathbb{V}$
  - restricted so that each key appears at most once.
  - $\mathbb{S}$ is the power set of $\mathbb{K}$.
  - $\mathbb{Z}_{\geq 0}$ denotes the positive integers.

Table Functions

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \emptyset \\
\texttt{size}(T) & : & \mathbb{T} \to \mathbb{Z}_{\geq 0} & = |T| \\
\texttt{singleton}(k, v) & : & \mathbb{K} \times \mathbb{V} \to \mathbb{T} & = \{(k, v)\} \\
\texttt{filter}(f, T) & : & ((\mathbb{V} \to \{\texttt{T}, \texttt{F}\}) \times \mathbb{T}) & \\
 & & \quad \to \mathbb{T} & = \{(k, v) \in T \mid f(v)\} \\
\texttt{map}(f, T) & : & ((\mathbb{K} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T}) & \\
 & & \quad \to \mathbb{T} & = \{(k, f(k, v)) \mid ((k, v) \in T)\} \\
\texttt{insert}(f, T, (k, v)) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} & \\
 & & \quad \times (\mathbb{K} \times \mathbb{V}) \to \mathbb{T} & = \\
 & & & \begin{cases} (T \setminus \{(k, v)\}) \cup \\ \qquad \{(k, f(v, v'))\} \\ \qquad \text{if } (k, v') \in T \\ T \cup \{(k, v)\} \\ \qquad \text{otherwise} \end{cases} \\
\texttt{delete}(T, k)) & : & \mathbb{T} \times \mathbb{K} \to \mathbb{T} & = \{(k', v) \in T | k \neq k'\}
\end{array}
$$

Table Functions

$$
\begin{array}{lcll}
\texttt{find}(T,k) & : & \mathbb{T}\times\mathbb{K}\to(\mathbb{V}\cup\bot) & = \begin{cases} v & (k,v)\in T \\ \bot & \textit{otherwise} \end{cases} \\
\texttt{merge}(f,T_1,T_2) & : & (\mathbb{V}\times\mathbb{V}\to\mathbb{V})\times\mathbb{T}\times\mathbb{T}\to\mathbb{T} & = \\
 & & \multicolumn{2}{l}{\bigcup_{k\in\mathbb{K}} \begin{cases} \{(k,f(v_1,v_2))\} & (k,v_1)\in T_1 \wedge (k,v_2)\in T_2 \\ \{(k,v_1)\} & (k,v_1)\in T_1 \wedge (k,v_2)\notin T_2 \\ \{(k,v_2)\} & (k,v_2)\in T_2 \wedge (k,v_1)\notin T_1 \end{cases}} \\
\texttt{extract}(T,S) & : & \mathbb{T}\times\mathbb{S}\to\mathbb{T} & = \{(k,v)\in T | k\in S\} \\
\texttt{erase}(T,S) & : & \mathbb{T}\times\mathbb{S}\to\mathbb{T} & = \{(k,v)\in T | k\notin S\}
\end{array}
$$

Table Examples

- Suppose we have the two tables:
  - $Summer = \{tree \mapsto green,\ sky \mapsto blue,\ cmuq \mapsto tan\}$
  - $Fall = \{grass \mapsto gray,\ tree \mapsto brown\}$

- `merge` $(fn\ (a, b) \Rightarrow b)\ Summer\ Fall$
  - $\{grass \mapsto gray,\ tree \mapsto brown,\ sky \mapsto blue,\ cmuq \mapsto tan\}$

Table Examples

- Suppose we have the two tables:
  - $Summer = \{tree \mapsto green,\ sky \mapsto blue,\ cmuq \mapsto tan\}$
  - $Fall = \{grass \mapsto gray,\ tree \mapsto brown\}$

- `extract`$(Summer, \{sky,\ grass\})$
  - $\{sky \mapsto\ blue\}$

Table Examples

- Suppose we have the two tables:
  - $Summer = \{tree \mapsto green,\ sky \mapsto blue,\ cmuq \mapsto tan\}$
  - $Fall = \{grass \mapsto gray,\ tree \mapsto brown\}$

- `erase`$(Summer, \{sky,\ grass\})$
  - $\{tree \mapsto green,\ cmuq \mapsto tan\}$

Table Examples

- Other useful functions from the library
- `collect`:$(key \times \alpha)\ seq \to (\alpha\ seq)\ table$
- `fromSeq`: $(key \times \alpha)\ seq \to \alpha\ table$
  - `fromSeq`$(A) = \{k \mapsto s_0 : (k \mapsto S) \in collect(A)\}$

Table Functions

- Major differences from sets:
  - `find` returns the value if key is in the table else returns $\perp$ (`NONE`).
  - `insert/merge` need a function to combine if the key is already in the/both table(s).
- Just as with sets, there is symmetry between
  - `extract` and `find`
  - `merge` and `insert`
  - `erase` and `delete`

Cost Models for Tables

	Work	*Span*
`size`(T) `singleton`(k, v)	$O(1)$	$O(1)$
`filter`(f, T)	$O\left(\sum_{(k,v)\in T} W(f(v))\right)$	$O\left(\log \lvert T\rvert + \max_{(k,v)\in T} S(f(v))\right)$
`map`(f, T)	$O\left(\sum_{(k,v)\in T} W(f(k,v))\right)$	$O\left(\max_{(k,v)\in T} S(f(k,v))\right)$

Cost Models for Tables

	Work	*Span*
`find`(S, k) `insert`$(T, (k, v))$ `delete`(T, k)	$O(W_c \log \lvert T \rvert)$	$O(S_c \log \lvert T \rvert)$
`extract`(T_1, T_2) `merge`(T_1, T_2) `erase`(T_1, T_2)	$O\left(W_c m \log(1 + \frac{n}{m})\right)$	$O\left(S_c \log(n+m)\right)$

$n = \max(\lvert T_1 \rvert, \lvert T_2 \rvert)$ $\quad$ $m = \min(\lvert T_1 \rvert, \lvert T_2 \rvert)$

Summary

- Collect
- Map-Collect-Reduce
- Sets
- Tables