Parallel Lookup Functions for High-Performance Data Exploration

Connor Lien, Dixin Tang, Aditya Parameswaran

EPIC DATA lab
UC Berkeley

Background

Lookup functions are extremely popular in spreadsheet data analysis

- Incredibly intuitive and common in Excel and Google Sheets
- Can powerfully help users synthesize data

Lookup functions scale extremely poorly

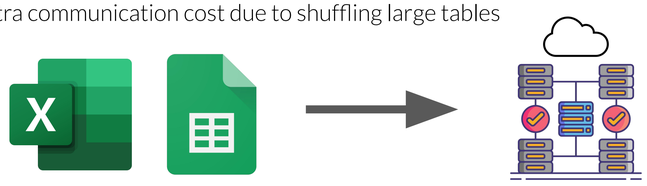
- Excel and Google Sheets take minutes on large inputs, and fail to handle cases with many rows which limits the function's use case
- Currently no APIs that support fast lookups for tabular data

Idea: scale lookup functions using **distributed algorithms** based on database joins and well-known approximate search algorithms

- Discuss the tradeoffs of each in terms of CPU cost, network cost, and implementation simplicity

Unique Challenges

- Must preserve order — track with an index column
- Combining subresults of approximate (fuzzy) matching
- Extra communication cost due to shuffling large tables



VLOOKUP Formula

We focus on a common lookup, VLOOKUP. Other types of lookup (e.g. HLOOKUP, XLOOKUP) are very similar but have different APIs.

```
VLOOKUP(A6, $B$1:$C$10, 2, TRUE)
```

- The "values" we are looking for — Can be constant or a vector
- The **constant** "table" we are operating in — The leftmost column is the **search range**
- The "col_idxes", the column we should fetch the results from — Usually **constant**, but can also be a vector
- Whether or not the operation is an **approximate** (fuzzy) lookup

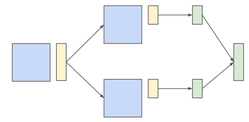
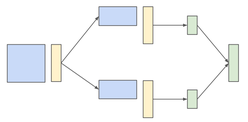
Example execution (of the above formula)

	A	B	C
1	Alexander	Amelia	0
2	Benjamin	Brooklyn	1
3	Carter	Charlotte	2
4	Daniel	Delilah	3
5	Elijah	Emma	4
6	Finn	Freya	5
7	Grayson	Gianna	6
8	Henry	Harper	7
9	Isaac	Isabella	8
10	James	Josephine	9

1. Find the value **(Finn)** we are looking for in search range
   a. For approximate, pick the position before the index it should be inserted into **(before Freya, row 5)**
2. With the search range as index 1, find the column we are retrieving results from (the numerical column)
3. Result is **4**

Distributed VLOOKUP

Approximate (Fuzzy) Match

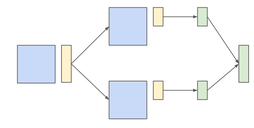
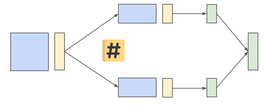
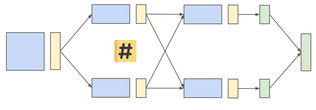
Broadcast table, block values		Good for large values and small table or if you have a fast network. Easiest to implement.
Broadcast values, block table		Sacrificing linear time passes over values and col_idxes to search over a smaller amount of the table
Locally range partition values, block table		Good if data takes up a lot of memory and/or if network is slow. Uses minimal network cost but still achieves lookup parallelism.
Distributed range partition		Good balance of network cost and CPU cost, but range partitioning blocks and requires worker synchronization

Combining approximate subresults (correctness lemma)

Having range partitioned values and a block partitioned search range will always produce a correct distributed result by combining local approximate VLOOKUPs

- Two cases: value is found within the range, or it is not (approximate match). If it's found, it is correct.
- If it's not found, the edge cases are when the desired value sits at edges of the blocked search range.
  - If there are duplicates between blocks, the value will always be partitioned into the worker with the lower index, and since VLOOKUP approximates before, it will find the correct value.

Exact Match

Broadcast table, block values		Good for large values and small table or if you have a fast network
Locally hash values and table		Sacrificing linear time passes over values and col_idxes to search over a smaller amount of the table
Distributed hash partition		Good if data takes up a lot of memory and/or if network is slow. Uses minimal network cost but still achieves lookup parallelism.
Distributed range partition		Only useful if you know the rough distribution of data, or use a quantile estimation algorithm
Symmetric hash lookup		(Future work) could be used for streaming lookup

Implementation + Evaluation

Case Study: Dataframes

- Lookup functions using **pandas DataFrames**
  - Why pandas? It's convenient, fast, and has a built-in index to track order (though sorting index can be slow)
- Parallel execution achieved using **Dask**

High-Level API

- Contains branching logic for match type, and uses fastest implementation depending on input data types
  - Slightly faster implementation for numerical inputs
- Performs **column compaction**
  - Filter out unused columns to reduce network and memory cost
- Constants optimization
  - If constants are inputted for values and col_idxes, does a local search and then replicates the result

Distributed Method Used

- **Distributed range partition** for approximate match
- **Distributed hash partition** for exact match

Benchmarking

- Randomly generated and sampled string data
- Run on AWS EC2 c5.4xlarge instances
- Total time starts and ends all data on a single node
- Execution time only includes lookup execution
- ~80% execution scaling with 8 cores for approximate match
- ~55% execution scaling with 8 cores for exact match