COMP 430
Intro. to Database Systems
NoSQL intro & MapReduce
What’s wrong with using SQL & RDBMs?

• Programmer must learn another language.
• Programmer must learn another way to organize data.
• Application must map between two data representations.
• Schema is difficult to change.

• SQL is clunky.
• SQL doesn’t deal well with super/sub-classes.
• SQL doesn’t deal well with unstructured data.
What is NoSQL?

Disparate DBs & data-manipulation S/W not using SQL

Typically:
• Library that plugs into many high-level languages
• Use same language as application
• Flexible data representation
• Use same data representation as application

Frequently:
• Grab-bag of data-manipulation functions, including filters and joins
MapReduce
Map and Reduce as traditional primitives

\[ f: X \rightarrow Y \]
\[ \text{map}(f, [x_1, ..., x_n]) = [f(x_1), ..., f(x_n)] \]

\[ g: X \rightarrow X \]
\[ \text{reduce}(g, \text{base}, [x_1, ..., x_n]) = g(...g(g(x_1, \text{base}), x_2), x_3), ..., x_n) \]

G usually associative & commutative, so that grouping and order on RHS is irrelevant.
MapReduce ≈ map + reduce + ...

Class.map: K V → [(K',V')]

All data are key/value pairs.

shuffle: [[(K',V')]] → [K',[V']]]

Can map each value to many.

Regroups so each key maps to a list of its values.

Class.reduce: K' [V'] → (K',V'')

Instead of fn args, those fns are now implicitly in each Class’ own map & reduce.

MapReduce(Class, input_kv_list) =

kv_list_list = [Class.map(input_kv) for input_kv in input_kv_list]
k_vlist_list = shuffle(kv_list_list)
output_kv_list = [Class.reduce(key,vlist) for (key,vlist) in k_vlist_list]
return output_kv_list
method map(id, content):
    for word in content:
        emit(word, 1)

method reduce(word, counts):
    emit(word, sum(counts))
MapReduce ≈ ... + parallel implementation

Programmer doesn’t have to think about parallel implementation.

• map – embarrassingly parallel
• shuffle – encapsulates most of the parallel communication
• reduce – easily parallelized as a tree

• Can intermingle stages – no barrier needed.
• Can separately load-balance map & reduce stages.
• Can restart or duplicate tasks for fault-tolerance.
Examples
method map(id, content):
    for word in content:
        emit(word, 1)
method reduce(word, counts):
    emit(word, sum(counts))

method map(id, content):
    counts = {}
    for word in content:
        counts[word] += 1
    for word in counts:
        emit(word, counts[word])
method reduce(word, counts):
    emit(word, sum(counts))

Each file contains many words. Lots of communication in shuffle.

Reduce some during map stage.

Idea generalized by combiner.
Average price per item for each customer

To compute average, need total price & #items.

```python
method map(customer_id, purchase_info):
    for item in purchase_info.items:
        emit(customer_id, {total_price: item.quantity * item.price,
                           item_count: item.quantity})

method reduce(customer_id, info_list):
    totals, counts = unzip(info_list)
    emit(customer_id, {total_price: sum(totals),
                       item_count: sum(counts)})
```

A tuple with
- Date, location, ...
- List of items, their quantities, and prices

Can use a finisher to compute averages at the end.
Join (reduce-side)

Equi-join on ‘key’.

```python
method map(table_name, record):
    emit(record.key, {table_name, record})

method reduce(keyvalue, table_records):
    for table_name1, record1 in table_records:
        for table_name2, record2 in table_records:
            if (table_name1 != table_name2):
                emit(keyvalue, join(record1, record2))
```

For tables with $m, n$ records, makes $(mn)^2$ comparisons!
Some other practical concerns
MapReduce inefficient(?)

• General framework
  • Could write better algorithms directly.
  • Generally no supplemental data – e.g., indices for joins.
  • Often good enough, and greatly simplifies programming.

• Have to parse data at each use.
  • Complex data stored on disk as data, XML, JSON, ... instead of simple tables
MapReduce as low-level primitive

• Coming up with appropriate map & reduce can be difficult.

• Joins, aggregation, ... often provided as higher-level primitives, with MapReduce as the underlying implementation.