MapReduce & HyperDex

Kathleen Durant PhD
Lecture 21 CS 3200
Northeastern University
Distributing Processing Mantra

• Scale “out,” not “up.”
• Assume failures are common.
• Move processing to the data.
• Process data sequentially and avoid random access.
• Hide system-level details from the application developer.
• Incorporate seamless scalability.
Drivers to MapReduce

• Our ability to store data is fast overwhelming our ability to process what we store
  • So you can write it you just can’t use it for any calculations
• Increases in capacity are outpacing improvements in bandwidth
  • So you can write it you just can’t read it back in a reasonable time
Introduction to Parallelization

- Writing algorithms for a cluster
  - On the order of 10,000 or more machines
  - Failure or crash is not an exception, but common phenomenon
  - Parallelize computation
  - Distribute data
  - Balance load
- Makes implementation of conceptually straightforward computations challenging
MapReduce

- Wanted: A model to express computation while hiding the messy details of the execution
- Inspired by map and reduce primitives in functional programming
  - Apply map to each input record to create a set of intermediate key-value pairs
  - Apply reduce to all values that share the same key (like GROUP BY)
- Automatically parallelized
- Re-execution as primary mechanism for fault tolerance
What is MapReduce?

• Programming model for expressing distributed computations on massive amounts of data
  
  AND

• An execution framework for large-scale data processing on clusters of commodity servers
Typical MapReduce Application

**MAP**
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results

**REDUCE**
- Aggregate intermediate results
- Generate final outcome
Programming Model

• Transforms set of input key-value pairs to set of output key-value pairs
  • Map function written by user
  • Map: \((k_1, v_1) \rightarrow \text{list } (k_2, v_2)\)
  • MapReduce library groups all intermediate pairs with same key together

• Reduce written by user
  • Reduce: \((k_2, \text{list } (v_2)) \rightarrow \text{list } (v_2)\)
  • Usually zero or one output value per group
  • Intermediate values supplied via iterator (to handle lists that do not fit in memory)
Execution Framework

- Handles scheduling of the tasks
  - Assigns workers to maps and reduce tasks
  - Handles data distribution
    - Moves the process to the data
  - Handles synchronization
    - Gathers, sorts and shuffles intermediate data
- Handles faults
  - Detects worker failures and restarts
- Understands the distributed file system
EXAMPLE: Count occurrences of each word in a document collection

Map( String key,  
    String value ):  
  // key: document name  
  // value: document contents  
  for each word w in value:  
    EmitIntermediate( w, "1" );

Reduce( String key,  
        Iterator values ):  
  // key: a word  
  // values: a list of counts  
  int result = 0;  
  for each v in values:  
    result += ParseInt( v );  
    Emit( AsString(result) );
Distributing work to nodes

• Focuses on large clusters
  • Relies on existence of reliable and highly available distributed file system
• Map invocations
  • Automatically partition input data into M chunks (16-64 MB typically)
  • Chunks processed in parallel
• Reduce invocations
  • Partition intermediate key space into R pieces, e.g., using hash(key) mod R
• Master node controls program execution
Dealing with failing nodes

- Master monitors tasks on mappers and reducers: idle, in progress, completed
- Worker failure (common)
  - Master pings workers periodically
  - No response => assumes worker failed
    - Resets worker’s map tasks, completed or in progress, to idle state (tasks now available for scheduling on other workers)
      - Completed tasks only on local disk, hence inaccessible
      - Same for reducer’s in-progress tasks
        - Completed tasks stored in global file system, hence accessible
  - Reducers notified about change of mapper assignment
- Master failure (unlikely)
  - Checkpointing or simply abort computation
Other considerations

- Conserve network bandwidth (“Locality optimization”)
  - Distributed file system assigns data chunks to local disks
  - Schedule map task on machine that already has a copy of the chunk, or one “nearby”
- Choose M and R much larger than number of worker machines
  - Load balancing and faster recovery (many small tasks from failed machine)
  - Limitation: $O(M+R)$ scheduling decisions and $O(M*R)$ in-memory state at master
  - Common choice: M so that chunk size is 16-64 MB, R a small multiple of number of workers
- Backup tasks to deal with machines that take unusually long for last few tasks
  - For in-progress tasks when MapReduce near completion
MapReduce

- Execution flow

Master

Intermediate keys partitioned into reduce *tasks*

Input files sent to map *tasks*

Input Files

Map workers

Reduce workers

Output
Map

• Interface
  • Input: <in_key, in_value> pair => <url, content>
  • Output: list of intermediate <key, value> pairs => list of <word, url>

Map Input: <url, content>

key = http://url0.com
value = “every happy family is alike.”

map()

<every, http://url0.com>
<happy, http://url0.com>
<family, http://url0.com>
...

key = http://url1.com
value = “every unhappy family is unhappy in its own way.”

map()

<every, http://url1.com>
<unhappy, http://url1.com>
<family, http://url1.com>
...

Map Output: list of <word, url>
Shuffle

- MapReduce system
  - Collects outputs from all *map* executions
  - Groups all intermediate values by the same key

Map Output: list of <word, url>

Reduce Input: <word, list of urls>
Reduce

• Interface
  • Input: <out_key, list of intermediate_value>
  • Output: <out_key, out_value>

```plaintext
Reduce Input: <word, list of urls>

Reduce Output: <word, string of urls>
```
Parallel Database

- SQL specifies what to compute, not how to do it
  - Perfect for parallel and distributed implementation
  - “Just” need an optimizer that can choose best plan in given parallel/distributed system
  - Cost estimate includes disk, CPU, and network cost
- Recent benchmarks show parallel DBMS can significantly outperform MapReduce
  - But many programmers prefer writing Map and Reduce in familiar PL (C++, Java)
- Recent trend: High-level PL for writing MapReduce programs with DBMS-inspired operators
SELECT Dim1, Dim2,
    SUM(Measure1) AS MSum,
    COUNT(*) AS RecordCount,
    AVG(Measure2) AS MAvg,
    MIN(Measure1) AS MMin,
    MAX(CASE
        WHEN Measure2 < 100
        THEN Measure2
        END) AS MMax
FROM DenormAggTable
WHERE (Filter1 IN ('A', 'B'))
    AND (Filter2 = 'C')
    AND (Filter3 > 123)
GROUP BY Dim1, Dim2
HAVING (MMin > 0)
ORDER BY RecordCount DESC
LIMIT 4, 8

db.runCommand(
    mapreduce: "DenormAggCollection",
    query: {
        filter1: [ '$in': ['A', 'B'] ],
        filter2: 'C',
        filter3: [ '$gt': 123 ]
    },
    map: function() { emit(
        { d1: this.Dim1, d2: this.Dim2 },
        { msum: this.measure1, recs: 1, mmin: this.measure1,
          mmax: this.measure2 < 100 ? this.measure2 : 0 }
    ),
    reduce: function(key, vals) {
        var ret = { msum: 0, recs: 0, mmin: 0, mmax: 0 }; for(var i = 0; i < vals.length; i++) {
            ret.msum += vals[i].msum;
            ret.recs += vals[i].recs;
            if(vals[i].mmin < ret.mmin) ret.mmin = vals[i].mmin;
            if((vals[i].mmax < 100) && (vals[i].mmax > ret.mmax))
                ret.mmax = vals[i].mmax;
        }
        return ret;
    },
    finalize: function(key, val) {
        val.mavg = val.msum / val.recs;
        return val;
    },
    out: 'result1',
    verbose: true
});
db.result1.
find({ mmin: { '$gt': 0 } }).
sort({ recs: -1 }).
skip(4).
limit(8);
My SQL vs. MongoDB

Database gurus have spoken out against MapReduce

Dave DeWitt, Michael Stonebraker

db.runCommand({ _mapreduce: "workouts", map: function () { emit( this.goalType, { '_cfcount': 1, 'distancekm_cfsum': isNaN(this.distancekm) ? null : this.distancekm, 'distancekm_cfname': isNaN(this.distancekm) ? 0 : 1, 'powerSongAlbum_cfcnt': (this.powerSongAlbum == null) ? 0 : 1, 'distancekm_cfname': isNaN(this.distancekm) ? null : this.distancekm, 'distancekm_cfname': isNaN(this.distancekm) ? null : this.distancekm } }); }, reduce: function (key,vals) { var ret = { 'distancekm_cfname': null, 'distancekm_cfsum': null, 'distancekm_cfname': 0, 'powerSongAlbum_cfcnt': 0, '_cfcount': 0 }; for(var i = 0; i < vals.length; i++) { var v = vals[i]; ret['distancekm_cfname'] += v['distancekm_cfname']; if(!isNaN(v['distancekm_cfname'])) ret['distancekm_cfname'] = (ret['distancekm_cfname'] == null) ? v['distancekm_cfname'] : (ret['distancekm_cfname'] > v['distancekm_cfname']) ? ret['distancekm_cfname'] : v['distancekm_cfname']; ret['_cfcount'] += v['_cfcount']; if(!isNaN(v['distancekm_cfname'])) ret['distancekm_cfsum'] = v['distancekm_cfsum'] + (ret['distancekm_cfsum'] == null ? 0 : ret['distancekm_cfsum']); } return ret; }, finalize: function (key,val) { return { 'totalkm': val['distancekm_cfsum'], 'workouts': val['_cfcount'], 'songcount': val['powerSongAlbum_cfcnt'], 'avgkm': (isNaN(val['distancekm_cfname']) || isNaN(val['distancekm_cfsum'])) ? null : val['distancekm_cfsum'] / val['distancekm_cfname'], 'maxkm': val['distancekm_cfname'], 'minkm': val['distancekm_cfname'] }; }, out: "s2mr", verbose: true });

MapReduce Summary

• MapReduce = programming model that hides details of parallelization, fault tolerance, locality optimization, and load balancing
• Simple model, but fits many common problems
• Implementation on cluster scales to 1000s of machines and more
  • Open source implementation, Hadoop, is available
• Parallel DBMS, SQL are more powerful than MapReduce and similarly allow automatic parallelization of “sequential code”
  • Never really achieved mainstream acceptance or broad open-source support like Hadoop
• Recent trend: simplify coding in MapReduce by using DBMS ideas
  • (Variants of) relational operators and BI being implemented on top of Hadoop