Big Data Processing: Map-Reduce & Parallel DBMS

Introduction to Databases
CompSci 316 Fall 2022
Announcements: Tues (Dec 6)

• LWOC – almost there!
• Transactions practice problem set coming soon
• See Announcements 2-5 on Ed
  • Practice problems
  • Important topics for final – final is comprehensive
  • Gradiance reopened for practice
• Course evaluations by 12/12!
  • Very important step to improve a class in the future
  • Your feedback during and after the semester matters
  • We need to hear from each of you
  • Small token of thanks (75% submissions- +2 to all in the final, 90% + 4)

• See the post on Project -- If you want to go for an in-class early presentation on Thursday, submit your current report and email Alex+me after the class cc-ing your group members
  • If many groups want to present, we may have to randomly select
Where are we now?

Relational model and queries
- Relational Model
- Query in SQL
- Query in RA

Database Design
- E/R diagram (design from scratch)
- Normal Forms (refine design)

Beyond Relational Model
- XML
- NOSQL JSON/MongoDB

Transactions
- Basics
- Concurrency Control
- Recovery

DBMS Internals and Query Processing
- Storage
- Index
- Join algo/Sorting
- Execution/Optimization

Advanced topics
- Recursion
- Big data processing: Map-Reduce Parallel DBMS

May/May not have time for data mining – there is a new course in Spring 2023: 590.04 on “Data Science” by Prof. Jian Pei if you are interested
One query/update
One machine

Multiple query/updates
One machine

Transactions

One query/update
Multiple machines

Parallel query processing
Map-Reduce, Spark, ..

Distributed query processing

Multiple query/updates, multiple machines:
Distributed transactions, Two-Phase Commit protocol, .. (not covered)
An overview of Map-Reduce
MapReduce: motivation

• Many problems can be processed in this pattern:
  • Given a lot of unsorted data
  • Map: extract something of interest from each record
  • Shuffle: group the intermediate results in some way
  • Reduce: further process (e.g., aggregate, summarize, analyze, transform) each group and write final results
  (Customize map and reduce for problem at hand)

☞ Make this pattern easy to program and efficient to run
  • Original Google paper in OSDI 2004
  • Hadoop is most popular open-source implementation
  • Spark still supports it
M/R programming model

- Input/output: each a collection of key/value pairs
- Programmer specifies two functions
  - `map(k_1, v_1) → list(k_2, v_2)`
    - Processes each input key/value pair, and produces a list of intermediate key/value pairs
  - `reduce(k_2, list(v_2)) → list(v_3)`
    - Processes all intermediate values associated with the same key, and produces a list of result values (usually just one for the key)
Simple Example: Map-Reduce

- Word counting
- Inverted indexes

Ack: Slide by Prof. Shivnath Babu
A similar M/R example: word count

• Expected input: a huge file (or collection of many files) with millions of lines of English text
• Expected output: list of (word, count) pairs
• Implementation
  • \( \text{map}(_, \text{line}) \rightarrow \text{list}(\text{word, count}) \)
    • Given a line, split it into words, and output \((w, 1)\) for each word \(w\) in the line
  • \( \text{reduce}(\text{word, list(count)}) \rightarrow (\text{word, count}) \)
    • Given a word \(w\) and list \(L\) of counts associated with it, compute \(s = \sum_{\text{count} \in L} \text{count}\) and output \((w, s)\)
• Optimization: before shuffling, map can pre-aggregate word counts locally so there is less data to be shuffled
  • This optimization can be implemented in Hadoop as a “combiner”
M/R execution

Reduce tasks:

Shuffle:

Map tasks:

Distributed file system

Data not necessary local

Distributed file system (e.g., HDFS)
M/R execution timeline

- When there are more tasks than workers, tasks execute in “waves”
  - Boundaries between waves are usually blurred
- Reduce tasks can’t start until all map tasks are done
Issues with M/R

• Numbers of map and reduce tasks
  • Larger is better for load balancing
  • But more tasks add overhead and communication

• Worker failure
  • Master pings workers periodically
  • If one is down, reassign its split/region to another worker

• “Straggler”: a machine that is exceptionally slow
  • Pre-emptively run the last few remaining tasks redundantly as backup
Why did we need a new programming model “Spark”?

• MapReduce greatly simplified big data analysis
• But as soon as it got popular, users wanted more:
  • More complex, multi-stage iterative applications (graph algorithms, machine learning)
  • More interactive ad-hoc queries
  • More real-time online processing
• All three of these apps require fast data sharing across parallel jobs
• Originally developed from Berkeley AMPLab – academic database research making huge impact!
Data Sharing in MapReduce

Slow due to replication, serialization, and disk IO
Data Sharing in Spark

10-100× faster than network and disk

In addition, stores all intermediate results and lineage as Resilient Distributed Datasets (RDDs) to avoid Recomputation from scratch after crashes.
Parallel Databases
Parallel processing

• Improve performance by executing multiple operations in parallel
• Cheaper to scale than relying on a single increasingly more powerful processor
Speedup

• Increase # processors → how much faster can we solve the same problem?
  • Overall problem size is fixed
Scaleup

- Increase # processors and problem size proportionally → can we solve bigger problems in the same time?
  - Per-processor problem size is fixed

![Graph showing linear scaleup and reality comparison](image-url)
Why linear speedup/scaleup is hard
Why linear speedup/scaleup is hard

• Startup
  • Overhead of starting useful work on many processors

• Communication
  • Cost of exchanging data/information among processors

• Interference
  • Contention for resources among processors

• Skew
  • Slowest processor becomes the bottleneck
Shared-nothing architecture

- Most scalable (vs. shared-memory and shared-disk)
  - Minimizes interference by minimizing resource sharing
  - Can use commodity hardware
- Also most difficult to program
Horizontal data partitioning

• Split a table $R$ into $p$ chunks, each stored at one of the $p$ processors

• Splitting strategies:
  • **Round robin or block-partitioning** distributes tuples arbitrarily but each processor gets the same amount of data (e.g., can assign the $i$-th row to chunk $(i \mod p)$)
  • **Hash-based partitioning on attribute $A$** assigns row $r$ to chunk $(h(r.A) \mod p)$
  • **Range-based partitioning on attribute $A$** partitioning the range of $R.A$ values into $p$ ranges, and assigns row $r$ to the chunk whose corresponding range contains $r.A$

Example: Suppose 3 machines, 9 tuples with single attribute $A$: 5, 9, 3, 6, 1, 7, 2, 8, 4
• **Round robin (arbitrary, equal load, one possibility):** (5, 6, 2) (9, 1, 8) (3, 7, 4)
  • What happens if we only need to access $A \leq 3$
• **Hash partitioning:** $A \% 3$ : (3, 6, 9) (1, 4, 7), (2, 5, 8)
  • What happens if all values are $\% 3 = 0$
• **Range partitioning:** $A \leq 3$, $3 < A < 7$, $A \geq 7$: (1, 2, 3), (4, 5, 6), (7, 8, 9)
  • What happens if all values are 9?
Practice Problem: Parallel DBMS
Example problem: Parallel DBMS

R(a,b) is horizontally partitioned across N = 3 machines.

Each machine locally stores approximately 1/N of the tuples in R.

The tuples are randomly organized across machines (i.e., R is block partitioned across machines).

Show a RA plan for this query and how it will be executed across the N = 3 machines.
Pick an efficient plan that leverages the parallelism as much as possible.

• SELECT a, max(b) as topb
• FROM R
• WHERE a > 0
• GROUP BY a
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
SELECT a, \text{max}(b) \text{ as top}_b \\
FROM R \\
WHERE a > 0 \\
GROUP BY a \\

If more than one relation on a machine, then “scan S”, “scan R” etc
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
Benefit of hash-partitioning

• What would change if we hash-partitioned R on R.a before executing the same query on the previous parallel DBMS and MR
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a

Prev: block-partition

1/3 of R
1/3 of R
1/3 of R

Machine 1
Machine 2
Machine 3
It would avoid the data re-shuffling phase
It would compute the aggregates locally
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
A brief summary of three approaches

• “DB”: parallel DBMS, e.g., Teradata
  • Same abstractions (relational data model, SQL, transactions) as a regular DBMS
  • Parallelization handled behind the scene, automatic optimizations
  • Transactions supported

• “BD (Big Data)” 10 years go: MapReduce, e.g., Hadoop
  • Easy scaling out (e.g., adding lots of commodity servers) and failure handling
  • Input/output in files, not tables
  • Parallelism exposed to programmers
  • Mostly manual optimization
  • No transactions/updates

• “BD” today: Spark
  • Compared to MapReduce: smarter memory usage, recovery, and optimization
  • Higher-level DB-like abstractions (but still no updates/transactions)
What are the “NOSQL” systems?

They have the ability to

• horizontally scale “simple read/write operations” throughput over many servers (e.g., joins are expensive or not supported)

• replicate and to distribute (partition) data over many servers

• a weaker concurrency model than ACID (BASE – Basically Available, Soft state, Eventually consistent)

• Efficiently use distributed indexes and RAM for data storage

• dynamically add new attributes to data records (like JSON)

• Example: MongoDB, CouchDB, Dynamo, MemBase…