Lecture 24: Parallel Databases

Wednesday, November 24, 2010

Overview

- Parallel architectures and operators: Ch. 20.1
- Map-reduce: Ch. 20.2
- Semijoin reductions, full reducers: Ch. 20.4
  - We covered this a few lectures ago

Parallel v.s. Distributed Databases

- Parallel database system:
  - Improve performance through parallel implementation

- Distributed database system:
  - Data is stored across several sites, each site managed by a DBMS capable of running independently

Parallel DBMSs

- Goal
  - Improve performance by executing multiple operations in parallel

- Key benefit
  - Cheaper to scale than relying on a single increasingly more powerful processor

- Key challenge
  - Ensure overhead and contention do not kill performance

Performance Metrics for Parallel DBMSs

- Speedup
  - More processors ➔ higher speed
  - Individual queries should run faster
  - Should do more transactions per second (TPS)
  - Fixed problem size overall, vary # of processors ("strong scaling")

- Scaleup
  - More processors ➔ can process more data
  - Fixed problem size per processor, vary # of processors ("weak scaling")
  - Batch scaleup
    - Same query on larger input data should take the same time
  - Transaction scaleup
    - N-times as many TPS on N-times larger database
    - But each transaction typically remains small

Linear v.s. Non-linear Speedup

- Speedup
- # processors (=P)
Linear v.s. Non-linear Scaleup

Batch Scaleup

\( \times 1 \) \( \times 5 \) \( \times 10 \) \( \times 15 \)

# processors (=P) AND data size

Challenges to Linear Speedup and Scaleup

- **Startup cost**
  - Cost of starting an operation on many processors

- **Interference**
  - Contention for resources between processors

- **Skew**
  - Slowest processor becomes the bottleneck

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Shared Memory

- Interconnection Network
- Global Shared Memory

Shared Disk

- Interconnection Network

Shared Nothing

- Interconnection Network
Shared Nothing

- Most scalable architecture
  - Minimizes interference by minimizing resource sharing
  - Can use commodity hardware
- Also most difficult to program and manage
- Processor = server = node
  - "Processor" ≠ core
- P = number of nodes

Question

- What exactly can we parallelize in a parallel DB?

Taxonomy for Parallel Query Evaluation

- Inter-query parallelism
  - Each query runs on one processor
- Inter-operator parallelism
  - A query runs on multiple processors
  - An operator runs on one processor
- Intra-operator parallelism
  - An operator runs on multiple processors

Horizontal Data Partitioning

- Relation R split into P chunks R₀, ..., Rₚ₋₁, stored at the P nodes
- Round robin: tuple tᵢ to chunk (i mod P)
- Hash based partitioning on attribute A:
  - Tuple t to chunk h(t.A) mod P
- Range based partitioning on attribute A:
  - Tuple t to chunk i if vᵢ₋₁ < t.A < vᵢ

Horizontal Data Partitioning

- All three choices are just special cases:
  - For each tuple, compute bin = f(t)
  - Different properties of the function f determine hash vs. range vs. round robin vs. anything

Parallel Selection

Compute σ₁₁⁺₁(R), or σ₁⁺₂⁺₂(R)

- On a conventional database: cost = B(R)
- Q: What is the cost on a parallel database with P processors?
  - Round robin
  - Hash partitioned
  - Range partitioned
Parallel Selection

• Q: What is the cost on a parallel database with P processors ?

• A: \( B(R) / P \) in all cases

• However, different processors do the work:
  – Round robin: all servers do the work
  – Hash: one server for \( \sigma_{sum}(R) \), all for \( \sigma_{B1 < A < B2}(R) \)
  – Range: one server only

Data Partitioning Revisited

What are the pros and cons ?

• Round robin
  – Good load balance but always needs to read all the data

• Hash based partitioning
  – Good load balance but works only for equality predicates and full scans

• Range based partitioning
  – Works well for range predicates but can suffer from data skew

Parallel Group By: \( \gamma_{A,\text{sum}(B)}(R) \)

• Step 1: server \( i \) partitions chunk \( R_i \) using a hash function \( h(t,A) \mod P: R_{i0}, R_{i1}, ..., R_{i,P-1} \)

• Step 2: server \( i \) sends partition \( R_{ij} \) to serve \( j \)

• Step 3: server \( j \) computes \( \gamma_{A,\text{sum}(B)} \) on \( R_{0j}, R_{1j}, ..., R_{P-1,j} \)

Cost of Parallel Group By

Recall conventional cost = \( 3B(R) \)

• Cost of Step 1: \( B(R)/P \) I/O operations

• Cost of Step 2: \( (P-1)/P \) blocks are sent
  – Network costs assumed to be much lower than I/O

• Cost of Step 3: \( 2B(R)/P \)
  – Why ?
  – When can we reduce it to 0 ?

Total = \( 3B(R) / P + \) communication costs

Parallel Group By: \( \gamma_{A,\text{sum}(B)}(R) \)

• Can we do better?

  Sum?
  Count?
  Avg?
  Max?
  Median?

Parallel Group By: \( \gamma_{A,\text{sum}(B)}(R) \)

• Sum(B) = \( \sum \) for \( (B_0, ..., B_n) \)

• Count(B) = \( \sum \) for \( B_i \) + ... + Count(B_n)

• Max(B) = \( \max \) for \( B_0, B_1, ..., B_n \)

• \( \text{Median}(B) = \) 
  – \( \text{distributive} \)
  – \( \text{algebraic} \)
  – \( \text{holistic} \)
Parallel Join: $R \bowtie_{A=B} S$

- Step 1:
  - For all servers in $[0,k]$, server $i$ partitions chunk $R_i$ using a hash function $h(t.A) \mod P$: $R_{i0}, R_{i1}, ..., R_{i,P-1}$
  - For all servers in $[k+1,P]$, server $j$ partitions chunk $S_j$ using a hash function $h(t.A) \mod P$: $S_{j0}, S_{j1}, ..., S_{j,P-1}$

- Step 2:
  - Server $i$ sends partition $R_iu$ to server $u$
  - Server $j$ sends partition $S_ju$ to server $u$

- Steps 3: Server $u$ computes the join of $R_{iu}$ with $S_{ju}$

Cost of Parallel Join

- Step 1: $(B(R) + B(S))/P$

- Step 2: $0$
  - $(P-1)/P$ blocks are sent, but we assume network costs to be $<<$ disk I/O costs

- Step 3:
  - $0$ if smaller table fits in main memory: $B(S)/p \leq M$
  - $4(B(R)+B(S))/P$ otherwise

Parallel Dataflow Implementation

- Use relational operators unchanged
- Add special split and merge operators
  - Handle data routing, buffering, and flow control
- Example: exchange operator
  - Inserted between consecutive operators in the query plan
  - Can act as either a producer or consumer
  - Producer pulls data from operator and sends to $n$ consumers
    - Consumer buffers input data from $n$ producers and makes it available to operator through getNext interface

Shared Nothing Parallel Databases

- Teradata
- Greenplum
- Netezza
- Aster Data Systems
- Datallegro
- Vertica
- MonetDB
  - Commercialized as Vectorwise

Example System: Teradata

Find all orders from today, along with the items ordered

```sql
SELECT * FROM Orders o, Lines i
WHERE o.item = i.item
AND o.date = today()
```
Example System: Teradata

```
select
  date = today()
join
scan Order o
hash
h(o.item)
AMP 4
AMP 5
AMP 6

select
  date = today()
join
scan Order o
hash
h(o.item)
AMP 1
AMP 2
AMP 3

select
  date = today()
join
scan Order o
hash
h(o.item)
AMP 4
AMP 5
AMP 6
```

Example System: Teradata

```
join
scan Item i
hash
h(i.item)
AMP 4
AMP 5
AMP 6

join
scan Item i
hash
h(i.item)
AMP 1
AMP 2
AMP 3

join
scan Item i
hash
h(i.item)
AMP 4
AMP 5
AMP 6
```

Example System: Teradata

```
contains all orders and all
lines where hash(item) = 1
contains all orders and all
lines where hash(item) = 2
contains all orders and all
lines where hash(item) = 3
```

MapReduce, Hadoop and Parallel Data Flow Systems

Parallel Join: $R \bowtie_X S$

```
Hash on X
Join each hash bucket
```

Parallel Group By: $\gamma_A \sum_B(R)$

```
Hash on A
sum(B) for each A-value
```
Parallel Duo–elim $\delta_R$

Hash tuple

Remove duplicates

MapReduce Programming Model

- **Input & Output:** each a set of key/value pairs
- **Programmer specifies two functions:**
  - `map` in $(key, value)$ -> list$(key, intermediate_value)$
  - `reduce` out$(key, list(intermediate_value))$ -> list$(output_value)$

Example: Document Processing

Abridged Declaration of Independence

A Declaration by the Representatives of the United States of America in Congress Assembled

...
Map Reduce

- Google: [Dean 2004]
- Open source implementation: Hadoop
- Map-reduce = high-level programming model and implementation for large-scale parallel data processing

MapReduce Programming Model

- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:
  - map (in_key, in_value) -> list(out_key, intermediate_value)
    - Processes input key/value pair
    - Produces set of intermediate pairs
  - reduce (out_key, list(intermediate_value)) -> list(out_value)
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values (usually just one)

*Inspired by primitives from functional programming languages such as Lisp, Scheme, and Haskell*
Implementation

- There is one master node
- Master partitions input file into \( M \) splits, by key
- Master assigns workers (=servers) to the \( M \) map tasks, keeps track of their progress
- Workers write their output to local disk, partition into \( R \) regions
- Master assigns workers to the \( R \) reduce tasks
- Reduce workers read regions from the map workers’ local disks

Interesting Implementation Details

- Worker failure:
  - Master pings workers periodically,
  - If down then reassigns its splits to all other workers \( \rightarrow \) good load balance
- Choice of \( M \) and \( R \):
  - Larger is better for load balancing
  - Limitation: master needs \( O(M \times R) \) memory

Map-Reduce Summary

- Hides scheduling and parallelization details
- However, very limited queries
  - Difficult to write more complex tasks
  - Need multiple map-reduce operations
- Solution:
  - Use MapReduce as a runtime for higher level languages
  - Pig (Yahoo!), now apache project): RA-like operators
  - Hive (Facebook, now apache project): SQL
  - Scope (MS): SQL ! But proprietary...
  - DryadLINQ (MS): LINQ ! But also proprietary...

Interesting Implementation Details

Backup tasks:
- **Straggler** = a machine that takes unusually long time to complete one of the last tasks. Eg:
  - Bad disk forces frequent correctable errors (30MB/s \( \rightarrow \) 1MB/s)
  - The cluster scheduler has scheduled other tasks on that machine
- Stragglers are a main reason for slowdown
- Solution: pre-emptive backup execution of the last few remaining in-progress tasks

Isosurface Example
**Isosurface Example**

- Image of a 3D model with isosurface values.

**Example: Isosurface Extraction**

- Diagram showing the process of isosurface extraction.

**Example: Rendering**

- Diagram illustrating the rendering process.

**Why is MapReduce Successful?**

- **Easy**
  - Democratization of parallel computing
  - Just two serial functions
  - Time to first query: a few hours (contrast with parallel DB...)
- **Flexible**
  - Schema-free, “in situ” processing
  - “First, load your data into the database...”
  - “First, convert your images to bitmaps...”
  - “First, encode your 3D mesh as triangle soup...”
- **Fault-tolerance**

**What’s wrong with MapReduce?**

- Literally Map then Reduce and that’s it...
  - Realistic jobs have multiple steps
- What else?

**Realistic Job = Directed Acyclic Graph**

- Diagram showing a directed acyclic graph with inputs, outputs, channels, and processing vertices.
MapReduce Contemporaries

- Dryad (Microsoft)
  - Relational Algebra
- Pig (Yahoo)
  - Near Relational Algebra over MapReduce
- HIVE (Facebook)
  - SQL over MapReduce
- Cascading
  - Relational Algebra
- Clustera
  - U of Wisconsin
- Hbase
  - Indexing on HDFS

- RDBMS
  - Declarative query languages
  - Schemas
  - Logical Data Independence
  - Indexing
  - Algebraic Optimization
  - Caching/Materialized Views
  - ACID/Transactions

MapReduce vs RDBMS

- RDBMS
  - High Scalability
  - Fault-tolerance
  - "One-person deployment"

- MapReduce
  - Data Model
    - GPL
      - Typing (maybe)
  - Prog. Model
    - Workflow
      - Typing, provenance, scheduling, caching, task parallelism, reuse
    - Relations
      - Select, Project, Join, Aggregation, ...
      - Optimization, physical data independence, data parallelism, indexing
    - MapReduce
      - [(key, value)]
        - Map, Reduce
          - Massive data parallelism, fault tolerance
    - MS Dryad
      - IQueryable, IEnumerable
        - RA + Apply + Partitioning
          - Typing, massive data parallelism, fault tolerance
    - MPI
      - Arrays/Matrices
        - 70+ ops
          - Data parallelism, full control

Comparison

<table>
<thead>
<tr>
<th>Data Model</th>
<th>Prog. Model</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPL</td>
<td></td>
<td>Typing (maybe)</td>
</tr>
<tr>
<td>Workflow</td>
<td>dataflow</td>
<td>Typing, provenance, scheduling, caching, task parallelism, reuse</td>
</tr>
<tr>
<td>Relations</td>
<td>Select, Project, Join, Aggregation, ...</td>
<td>Optimization, physical data independence, data parallelism, indexing</td>
</tr>
<tr>
<td>MapReduce</td>
<td>[(key, value)]</td>
<td>Map, Reduce</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Massive data parallelism, fault tolerance</td>
</tr>
<tr>
<td>MS Dryad</td>
<td>IQueryable, IEnumerable</td>
<td>RA + Apply + Partitioning</td>
</tr>
<tr>
<td>MPI</td>
<td>Arrays/Matrices</td>
<td>70+ ops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data parallelism, full control</td>
</tr>
</tbody>
</table>