CompSci 516
Database Systems
Lecture 20
Parallel DBMS
Instructor: Sudeepa Roy
Announcements

• HW3 due on Monday, Nov 20, 11:55 pm (in 2 weeks)
  – See some clarifications on Piazza
Reading Material

• [RG]
  – Parallel DBMS: Chapter 22.1-22.5

• [GUW]
  – Parallel DBMS and map-reduce: Chapter 20.1-20.2

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Reading Material

• [RG]
  – Parallel DBMS: Chapter 22.1-22.5
  – Distributed DBMS: Chapter 22.6 – 22.14

• [GUW]
  – Parallel DBMS and map-reduce: Chapter 20.1-20.2
  – Distributed DBMS: Chapter 20.3, 20.4.1-20.4.2, 20.5-20.6

• Recommended readings:
  – Chapter 2 (Sections 1,2,3) of Mining of Massive Datasets, by Rajaraman and Ullman: http://i.stanford.edu/~ullman/mmds.html
  – Original Google MR paper by Jeff Dean and Sanjay Ghemawat, OSDI’ 04: http://research.google.com/archive/mapreduce.html

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Parallel and Distributed Data Processing

• Recall from Lecture 18!
• data and operation distribution if we have multiple machines

• Parallelism
  – performance

• Data distribution
  – increased availability, e.g. when a site goes down
  – distributed local access to data (e.g. an organization may have branches in several cities)
  – analysis of distributed data
# Parallel vs. Distributed DBMS

<table>
<thead>
<tr>
<th>Parallel DBMS</th>
<th>Distributed DBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Parallelization of various operations</td>
<td>• Data is physically stored across different sites</td>
</tr>
<tr>
<td>– e.g. loading data, building indexes, evaluating queries</td>
<td>– Each site is typically managed by an independent DBMS</td>
</tr>
<tr>
<td>• Data may or may not be distributed initially</td>
<td>• Location of data and autonomy of sites have an impact on Query opt., Conc. Control and recovery</td>
</tr>
<tr>
<td>• Distribution is governed by performance consideration</td>
<td>• Also governed by other factors:</td>
</tr>
<tr>
<td></td>
<td>– increased availability for system crash</td>
</tr>
<tr>
<td></td>
<td>– local ownership and access</td>
</tr>
</tbody>
</table>

Duke CS, Fall 2017  
CompSci 516: Database Systems  
Lecture 18
Parallel DBMS
Why Parallel Access To Data?

At 10 MB/s
1.2 days to scan

1,000 x parallel
1.5 minute to scan.

Parallelism:
divide a big problem
into many smaller ones
to be solved in parallel.
Parallel DBMS

- Parallelism is natural to DBMS processing
  - Pipeline parallelism: many machines each doing one step in a multi-step process.
  - Data-partitioned parallelism: many machines doing the same thing to different pieces of data.
  - Both are natural in DBMS!
DBMS: The parallel Success Story

• DBMSs are the most successful application of parallelism
  – Teradata (1979), Tandem (1974, later acquired by HP),...
  – Every major DBMS vendor has some parallel server

• Reasons for success:
  – Bulk-processing (= partition parallelism)
  – Natural pipelining
  – Inexpensive hardware can do the trick
  – Users/app-programmers don’t need to think in parallel
Some || Terminology

Ideal graphs

• Speed-Up
  – More resources means proportionally less time for given amount of data.

• Scale-Up
  – If resources increased in proportion to increase in data size, time is constant.
Some || Terminology

**In practice**

- **Due to overhead in parallel processing**

- **Start-up cost**
  
  Starting the operation on many processors, might need to distribute data

- **Interference**
  
  Different processors may compete for the same resources

- **Skew**
  
  The slowest processor (e.g. with a huge fraction of data) may become the bottleneck

---

**Ideal:** linear speed-up

**Actual:** sub-linear speed-up

---

**Ideal:** linear scale-up

**Actual:** sub-linear scale-up
Architecture for Parallel DBMS

• Among different computing units
  – Whether memory is shared
  – Whether disk is shared
Basics of Parallelism

• Units: a collection of processors
  – assume always have local cache
  – may or may not have local memory or disk (next)

• A communication facility to pass information among processors
  – a shared bus or a switch
Shared Memory

Interconnection Network

Global Shared Memory

shared memory
Shared Disk

Interconnection Network

local memory

shared disk
Shared Nothing

**Interconnection Network**

- local memory and disk
- no two CPU can access the same storage area
- all communication through a network connection
Architecture: At A Glance

**Shared Memory (SMP)**
- Easy to program
- Expensive to build
- Low communication overhead: shared mem.
- Difficult to scale up (memory contention)

**Shared Disk**
- Trade-off but still interference like shared-memory (contention of memory and nw bandwidth)

**Shared Nothing (network)**
- Hard to program and design parallel algos
- Cheap to build
- Easy to scale up and speed up
- Considered to be the best architecture

Sequent, SGI, Sun

VMScluster, Sysplex

Tandem, Teradata, SP2

we will assume shared nothing
What Systems Worked This Way

NOTE: (as of 9/1995)!

Shared Nothing
Teradata: 400 nodes
Tandem: 110 nodes
IBM / SP2 / DB2: 128 nodes
Informix/SP2: 48 nodes
ATT & Sybase: ? nodes

Shared Disk
Oracle: 170 nodes
DEC Rdb: 24 nodes

Shared Memory
Informix: 9 nodes
RedBrick: ? nodes
Different Types of DBMS Parallelism

• **Intra-operator parallelism**
  – get all machines working to compute a given operation (scan, sort, join)
  – OLAP (decision support)

• **Inter-operator parallelism**
  – each operator may run concurrently on a different site (exploits pipelining)
  – For both OLAP and OLTP

• **Inter-query parallelism**
  – different queries run on different sites
  – For OLTP

• **We’ll focus on intra-operator parallelism**
Data Partitioning

Horizontally Partitioning a table (why horizontal?):

**Range-partition**
- Good for equijoins, range queries, group-by
- Can lead to data skew

**Hash-partition**
- Good for equijoins
- But only if hashed on that attribute
- Can lead to data skew

**Block-partition or Round Robin**
- Send i-th tuple to i-mod-n processor
- Good to spread load
- Good when the entire relation is accessed

Shared disk and memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning
Example

• $R(\text{Key}, A, B)$

• Can Block-partition be skewed?
  – no, uniform

• Can Hash-partition be skewed?
  – on the key: uniform with a good hash function
  – on A: may be skewed,
    • e.g. when all tuples have the same A-value
Parallelizing Sequential Evaluation Code

• “Streams” from different disks or the output of other operators
  – are “merged” as needed as input to some operator
  – are “split” as needed for subsequent parallel processing

• Different Split and merge operations appear in addition to relational operators

• No fixed formula for conversion

• Next: parallelizing individual operations
Parallel Scans

• Scan in parallel, and merge.
• Selection may not require all sites for range or hash partitioning
  – but may lead to skew
  – Suppose $\sigma_A = 10R$ and partitioned according to $A$
  – Then all tuples in the same partition/processor
• Indexes can be built at each partition
Parallel Sorting

Idea:

• Scan in parallel, and range-partition as you go
  – e.g. salary between 10 to 210, #processors = 20
  – salary in first processor: 10-20, second: 21-30, third: 31-40, ....

• As tuples come in, begin “local” sorting on each

• Resulting data is sorted, and range-partitioned

• Visit the processors in order to get a full sorted order

• Problem: skew!

• Solution: “sample” the data at start to determine partition points.
Parallel Joins

• Need to send the tuples that will join to the same machine
  – also for GROUP-BY

• Nested loop:
  – Each outer tuple must be compared with each inner tuple that might join
  – Easy for range partitioning on join cols, hard otherwise

• Sort-Merge:
  – Sorting gives range-partitioning
  – Merging partitioned tables is local
Parallel Hash Join

- In first phase, partitions get distributed to different sites:
  - A good hash function *automatically* distributes work evenly
- Do second phase at each site.
- Almost always the winner for equi-join
Dataflow Network for parallel Join

- Good use of split/merge makes it easier to build parallel versions of sequential join code.
Parallel Aggregates

• For each aggregate function, need a decomposition:
  – \( \text{count}(S) = \sum \text{count}(s(i)) \), ditto for \( \text{sum()} \)
  – \( \text{avg}(S) = \left( \sum \text{sum}(s(i)) \right) / \sum \text{count}(s(i)) \)
  – and so on...

• For group-by:
  – Sub-aggregate groups close to the source.
  – Pass each sub-aggregate to its group’s site.
    • Chosen via a hash fn.

Which SQL aggregate operators are not good for parallel execution?
Best serial plan may not be best

• Why?
• Trivial counter-example:
  – Table partitioned with local secondary index at two nodes
  – Range query: all of node 1 and 1% of node 2.
  – Node 1 should do a scan of its partition.
  – Node 2 should use secondary index.
Examples
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

We did this example for Map-Reduce in Lecture 12!
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
If more than one relation on a machine, then “scan S”, “scan R” etc

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

\[
\begin{align*}
\sigma_{a > 0} & \quad \text{scan} \\
\text{Machine 1} & \\
\text{1/3 of R} & \\
\end{align*}
\]

\[
\begin{align*}
\sigma_{a > 0} & \quad \text{scan} \\
\text{Machine 2} & \\
\text{1/3 of R} & \\
\end{align*}
\]

\[
\begin{align*}
\sigma_{a > 0} & \quad \text{scan} \\
\text{Machine 3} & \\
\text{1/3 of R} & \\
\end{align*}
\]

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

\[ \gamma_{a, \max(b)} \rightarrow b \]

\[ \sigma_{a > 0} \]

scan

Machine 1

1/3 of R

\[ \gamma_{a, \max(b)} \rightarrow b \]

\[ \sigma_{a > 0} \]

scan

Machine 2

1/3 of R

\[ \gamma_{a, \max(b)} \rightarrow b \]

\[ \sigma_{a > 0} \]

scan

Machine 3

1/3 of R

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

R(a, b)
SELECT a, max(b) as topb FROM R WHERE a > 0 GROUP BY a
\[
\gamma_{a, \max(b) \rightarrow \text{topb}}
\]

\[
\text{Hash on } a
\]

\[
\gamma_{a, \max(b) \rightarrow b}
\]

\[
\sigma_{a > 0}
\]

\[
\text{scan}
\]

\[
\text{Machine 1}
\]

\[
1/3 \text{ of } R
\]

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

• First Parallel DBMS
SELECT a, max(b) as topb FROM R
WHERE a > 0 GROUP BY a

Prev: block-partition
Hash-partition on a for R(a, b)

- 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
```
Hash-partition on a for R(a, b)

SELECT a, max(b) as topb FROM R WHERE a > 0 GROUP BY a
Benefit of hash-partitioning for Map-Reduce

• For MapReduce
  – Logically, MR won’t know that the data is hash-partitioned
  – MR treats map and reduce functions as black-boxes and does not perform any optimizations on them

• But, if a local combiner is used
  – Saves communication cost:
    • fewer tuples will be emitted by the map tasks
  – Saves computation cost in the reducers:
    • the reducers would have to do anything

SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
Column Store
(slides from Lecture 19)
Row vs. Column Store

• **Row store**
  – store all attributes of a tuple together
  – storage like “row-major order” in a matrix

• **Column store**
  – store all rows for an attribute (column) together
  – storage like “column-major order” in a matrix

• e.g.
  – MonetDB, Vertica (earlier, C-store), SAP/Sybase IQ,Google Bigtable (with column groups)
What is a column-store?

**row-store**

- easy to add/modify a record
- might read in unnecessary data

**column-store**

- only need to read in relevant data
- tuple writes require multiple accesses

$=>$ suitable for read-mostly, read-intensive, large data repositories

Ack: Slide from VLDB 2009 tutorial on Column store
Telco Data Warehousing example

Typical DW installation

Real-world example

"One Size Fits All? - Part 2: Benchmarking Results" Stonebraker et al. CIDR 2007

QUERY 2
SELECT account.account_number,
sum (usage.toll_airtime),
sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_id in ('AE', 'AA')
AND usage.toll_price > 0
AND source.type != 'CIBER'
AND toll.rating_method = 'IS'
AND usage.invoice_date = 20051013
GROUP BY account.account_number

<table>
<thead>
<tr>
<th></th>
<th>Column-store</th>
<th>Row-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>2.06</td>
<td>300</td>
</tr>
<tr>
<td>Query 2</td>
<td>2.20</td>
<td>300</td>
</tr>
<tr>
<td>Query 3</td>
<td>0.09</td>
<td>300</td>
</tr>
<tr>
<td>Query 4</td>
<td>5.24</td>
<td>300</td>
</tr>
<tr>
<td>Query 5</td>
<td>2.88</td>
<td>300</td>
</tr>
</tbody>
</table>

Why? Three main factors (next slides)

Ack: Slide from VLDB 2009 tutorial on Column store
Telco example explained (1/3): read efficiency

row store

read pages containing entire rows

one row = 212 columns!

is this typical? (it depends)

What about vertical partitioning? (it does not work with ad-hoc queries)

column store

read only columns needed

in this example: 7 columns

caveats:
- “select *” not any faster
- clever disk prefetching
- clever tuple reconstruction

Ack: Slide from VLDB 2009 tutorial on Column store
Telco example explained (2/3): *compression efficiency*

1. Columns compress better than rows
   1. Typical row-store compression ratio 1:3
   1. Column-store 1:10

1. **Why?**
   1. Rows contain values from different domains
      => more entropy, difficult to dense-pack
   1. Columns exhibit significantly less entropy
   1. **Examples:**
      
      Male, Female, Female, Female, Male

1. Caveat: CPU cost (use lightweight compression)

Ack: Slide from VLDB 2009 tutorial on Column store
Telco example explained (3/3): sorting & indexing efficiency

1. Compression and dense-packing free up space
   1. Use multiple overlapping column collections
   1. Sorted columns compress better
   1. Range queries are faster
   1. Use sparse clustered indexes

Ack: Slide from VLDB 2009 tutorial on Column store