CS-460 Database Management Systems

(Parallel Databases)

Haridimos Kondylakis

Computer Science Department, University of Crete
Todays Contents

1. Introduction
2. Parallel Databases
Big data: Through the eyes of computation

✓ Computer science is the topic about

*the computation of function f(x)*

✓ Big data: the data parameter \( x \) is horrendously large: PB or EB

What is the challenge introduced to query answering?

Fallacies:

✓ *Big data introduces no fundamental problems*
✓ *Big data = MapReduce (Hadoop)*
✓ *Big data = data quantity (scalability)*

Are these true?
Flashback: Relational queries

Questions:
• What is a relational schema? A relation? A relational database?
• What is a query? What is relational algebra?
• What does relationally completeness mean?
• What is a conjunctive query?
Traditional database management systems

• A **database** is a collection of data, typically containing the information about one or more related organizations.

• A **database management system (DBMS)** is a software package designed to store and manage databases.

  • Database: local
  • DBMS: centralized; single processor (CPU); managing local databases (single memory, disk)
Facebook: Graph Search

✓ Find me restaurants in New York my friends have been to in 2013
  • friend(pid1, pid2)
  • person(pid, name, city)
  • dine(pid, rid, dd, mm, yy)

✓ SQL query (in fact, a conjunctive query, or an SPC query)
  select rid
  from friend(pid1, pid2), person(pid, name, city),
       dine(pid, rid, dd, mm, yy)
  where pid1 = p0 and pid2 = person.pid and
       pid2 = dine.pid and

Facebook: more than 1.38 billion nodes, and over 140 billion links

Is it feasible on big data?
Example queries: Graph pattern matching

- Input: A pattern graph $Q$ and a graph $G$
- Output: All the matches of $Q$ in $G$, i.e., all subgraphs of $G$ that are isomorphic to $Q$

A bijective function $f$ on nodes:

$$(u, u') \in Q \iff (f(u), f(u')) \in G$$

- transportation network analysis
- Web site classification
- social position detection
- user targeted advertising
- knowledge base disambiguation …

What other graph queries do you know?
Graph pattern matching

Find all *matches* of a pattern in a graph

Identify suspects in a drug ring

Is this feasible?
Facebook: more than 1.38 billion nodes, and over 140 billion links

“Understanding the structure of drug trafficking organizations”
Querying big data: New challenges

Given a query $Q$ and a dataset $D$, compute $Q(D)$

What are new challenges introduced by querying big data?

- Does querying big data introduce new fundamental problems?
- What new methodology do we need to cope with the sheer size of big data $D$?

A departure from classical theory and traditional techniques
The good, the bad and the ugly

Traditional computational complexity theory of almost 50 years:
- The good: polynomial time computable (PTIME)
- The bad: NP-hard (intractable)
- The ugly: PSPACE-hard, EXPTIME-hard, undecidable…

What happens when it comes to big data?

Using SSD of 6G/s, a linear scan of a data set \( D \) would take
- 1.9 days when \( D \) is of 1PB (\( 10^{15} \)B)
- 5.28 years when \( D \) is of 1EB (\( 10^{18} \)B)

\( O(n) \) time is already beyond reach on big data in practice!

Polynomial time queries become intractable on big data!
Complexity classes within P

Polynomial time algorithms are no longer tractable on big data. So we may consider “smaller” complexity classes:

- **NC** (Nick’s class): highly parallel feasible
  - parallel polylog time
  - polynomially many processors

**Big open: P = NC?**

- **L**: O(log n) space
- **NL**: nondeterministic O(log n) space
- **polylog-space**: \( \log^k(n) \) space

\[ L \subseteq NL \subseteq \text{polylog-space} \subseteq P, \quad NC \subseteq P \]

Too restrictive to include practical queries feasible on big data
A class $Q$ of queries is **BD-tractable** if there exists a PTIME preprocessing function $\Pi$ such that

1. For any database $D$ on which queries of $Q$ are defined, $D' = \Pi(D)$
2. For all queries $Q$ in $Q$ defined on $D$, $Q(D)$ can be computed by evaluating $Q$ on $D'$ in parallel polylog time (NC)

Does it work? If a linear scan of $D$ could be done in $\log(|D|)$ time:

1. 15 seconds when $D$ is of 1 PB instead of 1.99 days
2. 18 seconds when $D$ is of 1 EB rather than 5.28 years

**BD-tractable queries are feasible on big data**
BD-tractable queries

A class $Q$ of queries is **BD-tractable** if there exists a PTIME preprocessing function $\Pi$ such that

- for any database $D$ on which queries of $Q$ are defined, $D' = \Pi(D)$
- for all queries $Q$ in $Q$ defined on $D$, $Q(D)$ can be computed by evaluating $Q$ on $D'$ in **parallel polylog time** (NC)

**Preprocessing: a common practice of database people**

- one-time process, offline, once for all queries in $Q$
- indices, compression, views, incremental computation, ...

**not necessarily reduce the size of $D$**

**$BDTQ_0$: the set of all BD-tractable query classes**
Fundamental problems for BD-tractability

**BD-tractable queries** help practitioners determine what query classes are tractable on big data.

No, a number of questions in connection with a complexity class!

- **Reductions**: how to transform a problem to another in the class that we know how to solve, and hence make it BD-tractable?
- **Complete problems**: Is there a natural problem (a class of queries) that is the hardest one in the complexity class? A problem to which all problems in the complexity class can be reduced.
- **How large is BDTQ₀?** Compared to P, NC?

Analogous to our familiar NP-complete problems

**Are we done yet?**

**Why do we need reduction?**

**Why do we care?**

Fundamental to any complexity classes: P, NP, …
Tractability revisited for big data

Yes, querying big data comes with new and hard fundamental problems.

BD-tractable queries: properly contained in $P$ unless $P = \text{NC}$.
What query classes are BD-tractable?

Boolean selection queries
✓ Input: A dataset D
✓ Query: Does there exist a tuple t in D such that t[A] = c?
Build a $B^+$-tree on the A-column values in D. Then all such selection queries can be answered in $O(\log(|D|))$ time.

Graph reachability queries
✓ Input: A directed graph G
✓ Query: Does there exist a path from node s to t in G?

What else?
Relational algebra + set recursion on ordered relational databases
D. Suciu and V. Tannen: A query language for NC, PODS 1994

Some natural query classes are BD-tractable
Challenges: query evaluation is costly

- Graph pattern matching by subgraph isomorphism
  - NP-complete to decide whether there exists a match
  - possibly exponentially many matches

- Membership problem for relational queries
  - Input: a query Q, a database D, and a tuple t
  - Question: Is t in Q(D)?
  - NP-complete if Q is a conjunctive query (SPC)
  - PSPACE-complete if Q is in relational algebra (SQL)

intractable even in the traditional complexity theory

Already beyond reach in practice when the data is not very big
Is it still feasible to query big data?

Can we do better if we are given more resources?
- Parallel and distributed query processing – TDD

Using 10000 SSD of 6G/s, a linear scan of $D$ might take:
- 1.9 days/10000 = 16 seconds when $D$ is of 1PB ($10^{15}$B)
- 5.28 years/10000 = 4.63 days when $D$ is of 1EB ($10^{18}$B)

Only ideally!

Yes, parallel query processing. But how?
Parallel Database Management Systems
The Solution – Parallel DBMS

- Increase the I/O bandwidth
  - Data partitioning
  - Parallel data access

- Origins (1980's): *database machines*
  - Hardware-oriented ⇒ bad cost-performance ⇒ failure
  - Notable exception: ICL's CAFS Intelligent Search Processor

- 1990's: same solution but using standard hardware components integrated in a multiprocessor
  - Software-oriented
  - Standard essential to exploit continuing technology improvements
Multiprocessor Objectives

• High-performance with better cost-performance than mainframe or vector supercomputer
• Use many nodes, each with good cost-performance, communicating through network
  • Good cost via high-volume components
  • Good performance via bandwidth
• Trends
  • Microprocessor and memory (DRAM): off-the-shelf
  • Network (multiprocessor edge): custom
• The real challenge is to parallelize applications to run with good load balancing
Data Server Architecture

Client

Application server

Data server

client interface
query parsing
data server interface
communication channel
application server interface
database functions

database
Objectives of Data Servers

- Avoid the shortcomings of the traditional DBMS approach
  - Centralization of data and application management
  - General-purpose OS (not DB-oriented)
- By separating the functions between
  - Application server (or host computer)
  - Data server (or database computer or back-end computer)
Data Server Approach: Assessment

- **Advantages**
  - Integrated data control by the server (black box)
  - Increased performance by dedicated system
  - Can better exploit parallelism
  - Fits well in distributed environments

- **Potential problems**
  - Communication overhead between application and data server
    - High-level interface
  - High cost with mainframe servers
Parallel Data Processing

- Three ways of exploiting high-performance multiprocessor systems:
  ① Automatically detect parallelism in sequential programs (e.g., Fortran, OPS5)
  ② Augment an existing language with parallel constructs (e.g., C*, Fortran90)
  ③ Offer a new language in which parallelism can be expressed or automatically inferred

- Critique
  ① Hard to develop parallelizing compilers, limited resulting speed-up
  ② Enables the programmer to express parallel computations but too low-level
  ③ Can combine the advantages of both (1) and (2)
Data-based Parallelism

• Inter-operation
  • $p$ operations of the same query in parallel

```
  op.3
 /     \
|      |
/      \   R
|      |
|      |
|      |
|      |
```

• Intra-operation
  → The same op in parallel
Parallel DBMS

• Loose definition: a DBMS implemented on a tightly coupled multiprocessor

• Alternative extremes
  • Straightforward porting of relational DBMS (the software vendor edge)
  • New hardware/software combination (the computer manufacturer edge)

• Naturally extends to distributed databases with one server per site
Parallel DBMS - Objectives

- Much better cost / performance than mainframe solution
- High-performance through parallelism
  - High throughput with inter-query parallelism
  - Low response time with intra-operation parallelism
- High availability and reliability by exploiting data replication
- Extensibility with the ideal goals
  - Linear speed-up
  - Linear scale-up
Performance of a database system

- **Throughput**: the number of tasks finished in a given time interval
- **Response time**: the amount of time to finish a single task from the time it is submitted

Can we do better given more resources (CPU, disk, ...)?

Parallel DBMS: exploring **parallelism**

- Divide a big problem into many smaller ones to be solved in parallel
- Improve performance

![Diagram of parallel DBMS and traditional DBMS](diagram.png)
Linear Speed-up

Linear increase in performance for a constant DB size and proportional increase of the system components (processor, memory, disk)
**Degree of parallelism -- speedup**

**Speedup**: for a given task, TS/TL,

- TS: time taken by a traditional DBMS
- TL: time taken by a parallel DBMS with more resources
- TS/TL: more sources mean proportionally less time for a task

- **Linear speedup**: the speedup is $N$ while the parallel system has $N$ times resources of the traditional system

**Question**: can we do better than linear speedup?
Linear Scale-up

Sustained performance for a linear increase of database size and proportional increase of the system components.
Degree of parallelism -- scaleup

**Scaleup**: TS/TL

- A task Q, and a task Qₙ, N times bigger than Q
- A DBMS Mₛ, and a parallel DBMS Mₗ, N times larger
- TS: time taken by Mₛ to execute Q
- TL: time taken by Mₗ to execute Qₙ

- **Linear scaleup**: if TL = TS, i.e., the time is constant if the resource increases in proportion to increase in problem size

**Question**: can we do better than linear speedup?
Why can’t it be better than linear scaleup/speedup?

- **Startup costs**: initializing each process
- **Interference**: competing for shared resources (network, disk, memory or even locks)
- **Skew**: it is difficult to divide a task into exactly equal-sized parts; the response time is determined by the largest part
- **Data partitioning and shipment costs**

*Question:* the more processors, the faster?

*How can we leverage multiple processors and improve speedup?*
Why parallel DBMS?

• Improve performance:

Almost died 20 years ago; with renewed interests because
  • Big data -- data collected from the Web
  • Decision support queries -- costly on large data
  • Hardware has become much cheaper

• Improve reliability and availability: when one processor goes down

Renewed interest: MapReduce
Parallel Database Management Systems

• Why parallel DBMS?
• Architectures
• Parallelism
  • Intraquery parallelism
  • Interquery parallelism
  • Intraoperation parallelism
  • Interoperation parallelism
Shared memory

A common memory

- **Efficient communication**: via data in memory, accessible by all
- **Not scalable**: shared memory and network become bottleneck -- interference; not scalable beyond 32 (or 64) processors
- Adding memory cache to each processor? Cache coherence problem when data is updated

Informix (9 nodes)

```
What is this?
```
Shared disk

- **Fault tolerance**: if a processor fails, the others can take over since the database is resident on disk
- **Scalability**: better than shared memory -- memory is no longer a bottleneck; but disk subsystem is a bottleneck
  - **Interference**: all I/O to go through a single network; not scalable beyond a couple of hundred processors

Oracle RDB (170 nodes)

```
M       M       M
|       |       |
P       P      P
```

**Interconnection network**

```
DB   DB   DB
```
Shared nothing

- **scalable:** only queries and result relations pass through the network
- **Communication costs and access to non-local disks:** sending data involves software interaction at both ends

Teradata: 400 nodes
IBM SP2/DB2: 128 nodes
Informix SP2: 48 nodes
Architectures of Parallel DBMS

Shared nothing, shared disk, shared memory

Tradeoffs of

- **Scalability**
- Communication speed
- Cache coherence

Shared-nothing has the best scalability
Pipelined parallelism

- The output of operation A is consumed by another operation B, before A has produced the entire output
  Many machines, each doing one step in a multi-step process

✓ Does not scale up well when:
  - the computation does not provide sufficiently long chain to provide a high degree of parallelism:
  - relational operators do not produce output until all inputs have been accessed – blocking, or
  - A’s computation cost is much higher than that of B
Data Partitioned parallelism

- Many machines performing the **same** operation on **different** pieces of data
  - Intraquery,
  - Interquery,
  - Intraoperation,
  - Interoperation

The parallelism behind MapReduce
Partitioning

Partition a relation and distribute it to different processors

• Maximize processing at each individual processor
• Minimize data shipping

Query types:

• scan a relation,
• point queries \((A = v)\),
• range queries \((v < A \text{ and } A < v')\)
Partitioning strategies

N disks, a relation R

• **Round-robin**: send the $j$-th tuple of R to the disk number $j \ mod \ n$
  - Even distribution: good for scanning
  - Not good for equal joins (point queries) and range queries (all disks have to be involved for the search)

• **Range partitioning**: partitioning attribute $A$, vector $[v_1, \ldots, v_{n-1}]$
  - send tuple $t$ to disk $j$ if $t[A]$ in $[v_{j-1}, v_j]$
  - good for point and range queries on partitioning attributes (using only a few disks, while leaving the others free)

  - **Execution skew**: distribution may not be even, and all operations occur in one or few partitions (scanning)

• **Hash partitioning**: hash function $f(t)$ in the range of $[0, n-1]$
  - Send tuple $t$ to disk $f(t)$
  - good for point queries on partitioning attributes, and sequential scanning if the hash function is even

  - No good for point queries on non-partitioning attributes and range queries
Replicated Data Partitioning

• High-availability requires data replication
  • simple solution is mirrored disks
    • hurts load balancing when one node fails
  • more elaborate solutions achieve load balancing
    • interleaved partitioning (Teradata)
    • chained partitioning (Gamma)
## Interleaved Partitioning

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary copy</td>
<td>$R_1$</td>
<td>$R_2$</td>
<td>$R_3$</td>
<td>$R_4$</td>
</tr>
<tr>
<td>Backup copy</td>
<td>$r_{2.3}$</td>
<td>$r_{1.1}$</td>
<td>$r_{1.2}$</td>
<td>$r_{1.3}$</td>
</tr>
<tr>
<td></td>
<td>$r_{3.2}$</td>
<td>$r_{3.2}$</td>
<td>$r_{2.1}$</td>
<td>$r_{2.2}$</td>
</tr>
</tbody>
</table>
## Chained Partitioning

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary copy</td>
<td>$R_1$</td>
<td>$R_2$</td>
<td>$R_3$</td>
<td>$R_4$</td>
</tr>
<tr>
<td>Backup copy</td>
<td>$r_4$</td>
<td>$r_1$</td>
<td>$r_2$</td>
<td>$r_3$</td>
</tr>
</tbody>
</table>
Placement Directory

• Performs two functions
  • $F_1$ (relname, placement attval) = lognode-id
  • $F_2$ (lognode-id) = phynode-id

• In either case, the data structure for $f_1$ and $f_2$ should be available when needed at each node
Interquery vs. intraquery parallelism

• **interquery:**
  • different queries or transactions execute in parallel
    • Easy: traditional DBMS tricks will do
    • Shared-nothing/disk: *cache coherence* problem
      Ensure that each processor has the latest version of the data in its buffer pool
      --flush updated pages to shared disk before releasing the lock

• **Intraquery:**
  • a *single* query in parallel on multiple processors
    • **Interoperation:** operator tree
    • **Intraoperation:** parallelize the same operation on different sets of the same relations
      • Parallel sorting
      • Parallel join
      • Selection, projection, aggregation
Relational operators

What are relational operators? Relationally complete?

• Projection: $\Pi_A R$
• Selection: $\sigma_C R$
• Join: $R_1 \bowtie_C R_2$
• Union: $R_1 \cup R_2$
• Set difference: $R_1 - R_2$

• Group by and aggregate (max, min, count, average)

How to support these operations in a parallel setting?
Intraoperation parallelism -- loading/projection

\[ \Pi_A R \], where \( R \) is partitioned across \( n \) processors

- Read tuples of \( R \) at all processors involved, in parallel
- Conduct projection on tuples
- Merge local results
  - Duplicate elimination: via sorting
Intraoperation parallelism -- selection

$$\sigma_C R$$, where $R$ is partitioned across $n$ processors

If $A$ is the partitioning attribute

- Point query: $C$ is $A = v$
  - a single processor that holds $A = v$ is involved
- Range query: $C$ is $v_1 < A$ and $A < v_2$
  - only processors whose partition overlaps with the range

If $A$ is not the partitioning attribute:

- Compute $\sigma_C R_i$ at each individual processor
- Merge local results

**Question**: evaluate $\sigma_{2 < A \text{ and } A < 6} R$, $R(A, B): \{(1, 2), (3, 4), (5, 6), (7, 2), (9, 3)\}$, and $R$ is range partitioned on $B$ to 3 processors
Intraoperation parallelism -- parallel sort

**sort R** on attribute A, where R is partitioned across n processors

If A is the partitioning attribute: **Range-partitioning**
- Sort each partition
- Concatenate the results

If A is not the partitioning attribute: **Range-partitioning sort**
- **Range partitioning** R based on A: redistribute the tuples in R
  - Every processor works in parallel: read tuples and send them to corresponding processors
- Each processor sorts its new partition locally when the tuples come in -- **data parallelism**
- Merge local results

Problem: **skew**
Solution: sample the data to determine the partitioning vector
Intraoperation parallelism -- parallel join

R1 $\bowtie_c$ R2

- Partitioned join: for equi-joins and natural joins
- Fragment-and replication join: inequality
- Partitioned parallel hash-join: equal or natural join
  - where R1, R2 are too large to fit in memory
  - Almost always the winner for equi-joins
Partitioned join

$R_1 \bowtie_{R_1.A = R_2.B} R_2$

- Partition $R_1$ and $R_2$ into $n$ partitions, by the same partitioning function in $R_1.A$ and $R_2.B$, via either
  - range partitioning, or
  - hash partitioning
- Compute $R^i_1 \bowtie_{R_1.A = R_2.B} R^i_2$ locally at processor $i$
- Merge the local results

**Question**: how to perform partitioned join on the following, with 2 processors?

- $R_1(A, B): \{(1, 2), (3, 4), (5, 6)\}$
- $R_2(B, C): \{(2, 3), \{3, 4\}\}$
Fragment and replicate join (broadcast)

\[ R_1 \bowtie_{R_1.A < R_2.B} R_2 \]

- Partition \( R_1 \) into \( n \) partitions, by any **partitioning method**, and distribute it across \( n \) processors
- Replicate the other relation \( R_2 \) across all processors
- Compute \( R_{j1} \bowtie_{R_1.A < R_2.B} R_2 \) locally at processor \( j \)
- Merge the local results

**Question**: how to perform fragment and replicate join on the following, with 2 processors?

- \( R_1(\text{A, B}): \{(1, 2), (3, 4), (5, 6)\} \)
- \( R_2(\text{B, C}): \{(2, 3), \{3, 4\}\} \)
Partitioned parallel hash join

R1 ⊙ R1.A = R2.B ⊙ R2, where R1, R2 are too large to fit in memory

• Hash partitioning R1 and R2 using hash function h1 on partitioning attributes A and B, leading to k partitions

• For i in [1, k], process the join of i-th partition R1^i ⊙ R2^i in turn, one by one in parallel
  • Hash partitioning R1^i using a second hash function h2, build in-memory hash table (assume R1 is smaller)
  • Hash partitioning R2^i using the same hash function h2
  • When R2 tuples arrive, do local join by probing the in-memory table of R1

Break a large join into smaller ones
Intraoperation parallelism -- aggregation

Aggregate on the attribute B of R, grouping on A

• decomposition
  • \text{count}(S) = \sum \text{count}(S_i); \text{ similarly for sum }
  • \text{avg}(S) = (\sum \text{sum}(S_i) / \sum \text{count}(S_i))

Strategy:
• Range partitioning R based on A: redistribute the tuples in R
• Each processor computes sub-aggregate -- data parallelism
• Merge local results

Alternatively:
• Each processor computes sub-aggregate -- data parallelism
• Range partitioning local results based on A: redistribute partial results
• Compose the local results
Describe a good processing strategy to parallelize the query:

```
select branch-name, avg(balance)
from account
group by branch-name
```

where the schema of account is

```
(account-id, branch-name, balance)
```

Assume that n processors are available
Describe a good processing strategy to parallelize the query:

```sql
select branch-name, avg(balance)
from account
group by branch-name
```

- Range or hash partition account by using branch-name as the partitioning attribute. This creates table `account_j` at each site `j`.
- At each site `j`, compute `sum(account_j) / count(account_j)`;
- Output `sum(account_j) / count(account_j)` – the union of these partial results is the final query answer
interoperation parallelism

Execute different operations in a single query in parallel
Consider R1 ⊗ R2 ⊗ R3 ⊗ R4

• Pipelined:
  • temp1 ← R1 ⊗ R2
  • temp2 ← R3 ⊗ temp1
  • result ← R4 ⊗ temp2

• Independent:
  • temp1 ← R1 ⊗ R2
  • temp2 ← R3 ⊗ R4
  • result ← temp1 ⊗ temp2 -- pipelining
Cost model

- Cost model: partitioning, skew, resource contention, scheduling
  - Partitioning: $T_{\text{part}}$
  - Cost of assembling local answers: $T_{\text{asm}}$
  - Skew: $\max(T_0, \ldots, T_n)$
  - Estimation: $T_{\text{part}} + T_{\text{asm}} + \max(T_0, \ldots, T_n)$
    May also include startup costs and contention for resources (in each $T_j$)

- Query optimization: find the “best” parallel query plan
  - Heuristic 1: parallelize all operations across all processors -- partitioning, cost estimation (Teradata)
  - Heuristic 2: best sequential plan, and parallelize operations -- partition, skew, ... (Volcano parallel machine)
Exercise: implement set difference

• Set difference: R1 \( - \) R2

• Develop a parallel algorithm that R1 and R2, computes R1 \( - \) R2, by using:
  • partitioned join
  • partitioned and replicated

• Questions: what can we do if the relations are too large to fit in memory?

Is it true that the more processors are used, the faster the computation is?
References

• M.T Oszu & P. Valduriez
  • Distributed DBMS

• Wenfei Fan
  • Lecture Notes on Research Topics in Distributed Databases