# Overview of Parallel DBMS

# Introduction

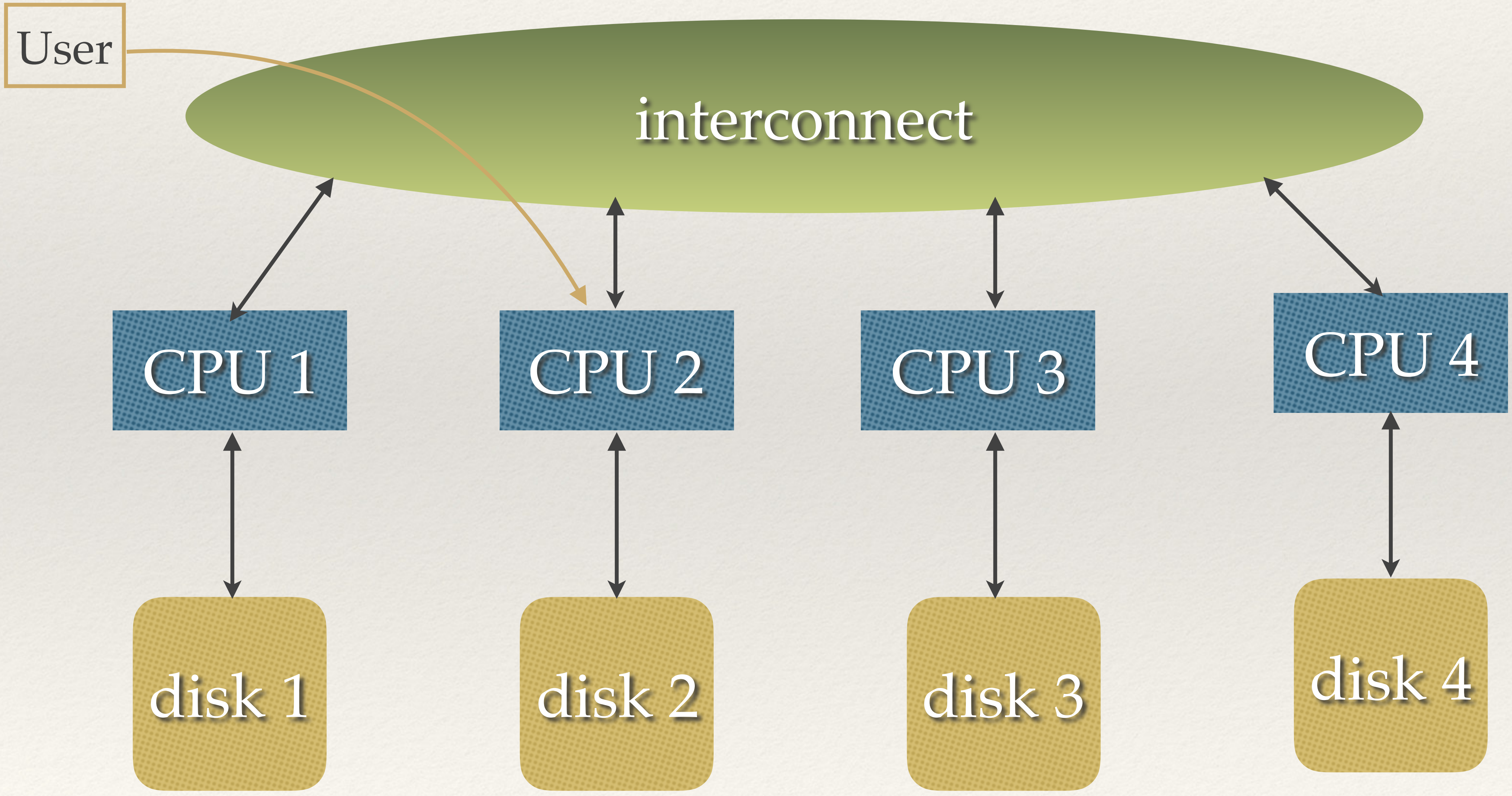
- Parallelism is everywhere
- Hardware vs. Software architecture
- Hardware
  - Multiple servers connected where each server comprises
    - several multi-core CPU’s
    - several “shared” disks (HDD/SSD)
    - shared memory

# Software Architecture

- Shared Memory
  - limited to one server
  - servers are now more powerful but still doesn't scale
- Shared Nothing: scalable and our focus
  - Pretend the disks and memory are divided up among the CPU (or CPU cores).
  - Works both within and across interconnected servers
  - For further discussion: the unit of parallelism is "Node" comprising CPU, memory and disk

# Pictorially

- User connects to one of the CPU's that acts as coordinator for that transaction (query)
- CPU 2 is the coordinator in the picture
- Some systems may have a separate coordinate node



# Compute vs Data Parallelism

- Compute parallelism: mostly scientific computing
  - small amount of shared data,  usu. memory resident
  - parallelism comes from doing different operations on the small dataset in parallel
- Data parallelism
  - doing operations in parallel on different chunks of data

# Two Kinds of Data Parallelism

- Within a transaction (i.e. within a query)
  - Parallelism speeds up processing
- Across transactions - multiple transactions accessing small amounts of data in parallel
  - Parallelism increases throughput

# Data Distribution

- To achieve data parallelism, the data has to be partitioned/ distributed across the nodes
- 3 methods
  - range partition on one or more column values
  - partition using hash of one or more column values
  - round-robin, divide data evenly across nodes
- Partitioning (distribution) key doesn't have to be same as clustered (primary) index key

# Range Partition

- Determine suitable key ranges for the distribution
- Ranges have to be known by all nodes (and coordinator)
- Advantage
  - equality search (on partitioning key) can be directed to one node
  - range searches on partitioning key need not use all nodes
- Disadvantage
  - hard to maintain ranges (skew can occur if not maintained)
  - unequal distribution of work for range (between) requests

# Hash Partition

- Fairly uniform distribution
- Skew can still occur if data is not somewhat uniform
- No need to maintain hash function (unlike range…)
  - no memory needed to store the ranges either
- equality search (on partitioning key) can be directed to one node
- disadvantage: range searches now have to go to all nodes
- most common method of distribution in PDBMS

# Round-Robin

- Basic idea is to evenly distribute rows across the nodes without using any keys for partitioning
- Great for quick loading of data
- Great if workload mostly consists of full table scans
- secondary indexes can still be built

# Indexing in Parallel DBMS

- Local structures are still traditional: B+ tree, Hash Index
- Access comprises the consideration of distribution key, if any, and the local index structure

# Example

- SELECT * FROM T1 WHERE c1 = 5;
- case 1: distribution key = index key
  - hash(c1 = 5) => determines node for c1=5
  - lookup the index on the node using c1 = 5
- case 2: distribution key <> index key
  - all nodes look for the value using index for c1 = 5

# Secondary Indexes

- Secondary Indexes are also tables (key, rowid) and can be distributed on the key value
- One approach:
  - for unique SI, distribute the SI table on the index key
  - for non-unique SI, build local SI table referring to local data (look up to secondary index go to all nodes)

# Unique SI lookup

- SELECT * FROM T1 WHERE c2 = 5
- assume c2 has a unique secondary index
- hash on c2 = 5, find the node
- look up the index for c2 = 5, find the row_id
- look up the node for row_id (in parallel DBMS, row_id would capture the node information)
- lookup the row on the node containing the row_id

# non-unique SI lookup

- SELECT * FROM t1 where c1 = 5
- c1 has a non-unique secondary index
- On all nodes simply look up the secondary index locally (just like the traditional method)

# Table Scans

- Except for index “directed” index lookup where request can go to a single node, most scan operations work in parallel on all nodes
- This is similar to regular processing

# Aggregations

- The basic aggregation algorithm remains the same
- Option 1
  - First aggregate all the rows local to the node
    - We would have (group by, aggregates) values
  - hash on the group by keys so that the equal group by value ends up on the same node
    - don't want to send all local values to same (coordinator) node …
  - "Globally" aggregate the locally aggregated values
    - count is now SUM of local count values
    - otherwise the same process
- If data is already distributed on the "Group By" key, then skip the local step

# Aggregations

- Option 2
  - Distribute all rows on the group by key
  - Do regular, viz. local, aggregation
- Option 2 better if number of groups is very large, e.g when local aggregation wouldn't result in much reduction in number of rows
- Typically used for "DISTINCT" computation but could be used to optimize "bad" (i.e. too many groups) aggregate cases

# Cross and non-equi-joins

- All rows of R have to be matched with all rows of S
- Duplicate R (or S) across all nodes
- Do nested loop join as usual

# Equijoin

- Two alternatives for data layout
  - Duplicate smaller of the R or S, leave other “as is”
  - distribute both by hashing (range partitioning could also be used) on the joining columns
- Any algorithms could then be used to join the two tables

# Parallel Hybrid Hash Join Algorithm

- Distribution of data can be combined with building/probing of the hash table to save on I/O
- Process 1: read and distributes the data by hashing on join columns
- Process 2: receives the rows and proceeds to build/probe the hash table (the overflow is processed just like the regular hybrid-hash join)

# Parallel Sorting

- Several purposes
  - collecting rows together for matching (join) and grouping (aggregation)
  - actual ordering - e.g. for order by
  - combination - order analytics
    - partition by - for matching
    - order by - for actual ordering

# Parallel Local Sorting

- Local-sort only: most common
  - sometimes it’s enough to simply sort data locally
  - e.g. for hash partitioned data, a local sort would be enough for merge join or for aggregation (or even for ordered analytics, if hash was on the “partition by” clause)
  - for order-by too, as data can be merged at the coordinator on the way out to the user
- Local sorting using external merge sort is enough

# Parallel Global Sorting

- Needed for two cases:
  - Ordered-analytics with one or very few partitions, i.e. no partition by clause or partition by has very few partitions
  - Large order-by where data may need to be exported in parallel

# Parallel Global Sort - Algorithm

- Sample the data set to determine the ranges for each node i.e. the range “splitting” vector
- Scan and distribute the data using the splitting vector
- Locally sort at each node
- Node 1 through Node N would have the globally sorted data in that order

# Ordered-Analytics

- If number of partitions in partition by is large
  - hash distribute on partition by
  - do local sort and processing
- If there is only one partition - consider Rank()
  - do global sort
  - will need to do additional accounting (e.g. count of the number of rows on preceding nodes)
  - scan and assign rank as preceding_count + local rank
- [General case for all functions is beyond the scope.]

# Skew Handling

- Hash distributed join processing is especially prone to skew as all matching rows have to be on same node
- Consider the worst case
  - all rows end up on the same node
  - follow the duplicate/"as is" method except we combine the "round-robin" distribution with the first step of hybrid-hash join
    - duplicate the smaller table and build hash table
    - "round-robin" the other table and probe the hash table (overflows stored locally)

# Query Optimization

- Need to consider data movement costs i.e. data distribution and layout
- Costing is now different (and we need to keep track of how the data is distributed)
- Need to handle skew
- Consider the alternatives in parallel algorithms, e.g. between 2 aggregation approaches, which two table distributions for joins etc.
- Basic optimization method still the same, just more complex