
Jens Teubner, DBIS Group
jens.teubner@cs.tu-dortmund.de

Winter 2015/16
Part IX

Parallel Databases
Motivation

It is increasingly attractive to leverage **parallelism** available in hardware.

**Reduced Cost:**
- Large monolithic systems are extremely complex to build.
- Smaller systems sell at much higher volumes, with much better price/performance ratio.

**Reduced Energy Consumption:**
- Performance scales linearly with clock frequency; energy consumption scales quadratically.
- Additional **cooling cost** makes this even worse.
- Modern chip designs are **power-limited** (俗称 multi-core)

**Prepare for Hardware Failures?**
- A spare COTS system is cheaper than a spare mainframe.
Scaling with Parallelism

Desirable: **speed-up** and **scale-up**

- **linear speed-up**
- **sub-linear speed-up**

- **linear scale-up**
- **sub-linear scale-up**
Different architectures have been proposed for parallel databases.
Advantages of shared memory architectures:
- Porting to shared memory architecture (relatively) easy.

Problems of shared memory architectures:
- **Contention** in interconnect
  - Here: *memory contention*
  - Hard to build scalable and fast interconnect.

  - **Interference**:  
    - Addl. CPUs *slow down* existing ones (e.g., due to contention).

  - Suitable for **low degrees of parallelism** (up to few tens).
Shared disk architectures have similar problems.

- contention and interference problems

Further:

- For read/write access, coherence tricky to get right.

- Shared nothing seems to be the method of choice.
Parallel Query Evaluation

**Intra-query parallelism:**

- **Pipeline parallelism:**
  - Assign plan operators to CPUs; send tuples from CPU to CPU.
  - Only works for **non-blocking operators**.
  - **Limited scalability**: few operators per plan; load balancing?

- **Data parallelism:**
Data Parallelism

Data parallelism goes particularly well with data partitioning.

→ Distribute tuples over nodes (→ horizontal partitioning)

→ Parallel scan; high I/O bandwidth

Round-Robin Partitioning:

- Easy, trivial load balancing

Range Partitioning:

- Need to access only those nodes that hold relevant data.
- Data skew may lead to trouble.
- May be beneficial for sorting, joining, etc.
- Range boundaries?
Hash Partitioning:

- **Data skew** less of a problem
- May also help certain operations (e.g., **joins**)
- **No knowledge** about data or types required
Parallelizing Operator Evaluation

**Scan:** Easy

→ Scan-heavy queries benefit easily from data parallelism.

**Sort:**

- Merge sort/external sort: run early stages in parallel, then merge
- With **range partitioning**, merging becomes trivial.
  → Thus, first range-partition (re-distribute) data, then sort.
  → Determine range boundaries with help of **sampling**.

**Join:**

- **Partition** (re-distribute) tuples (hash or range partitioning)
- $R_i \bowtie S_i$ joins can now be computed locally.
Parallel Joins (Using Merge Sort Locally)

- **input relation**
- **range partition**
- **local sort**
- **range partition and local sort**
- **input relation**
Parallel Joins (here: MPSM)

input relation

range partition

local sort

local sort

local sort

local sort

scan

scan

scan

scan

local sort

local sort

local sort

local sort

input relation
Instead: Sort, then Merge/Partition

- Re-distributes ("shuffles") likely limited by interconnect bandwidth.
- Perform merge/join during shuffle
  - Leverage available CPU capacity while I/O-limited.
Bloom filters\textsuperscript{23} can help reduce communication cost.

1. Partition and distribute outer join relation $R$.
2. On each node $H_i$, compute Bloom filter vector for $R_i$.
3. Broadcast all Bloom filters to all nodes.
4. Partition and distribute $S$, but filter tuples before sending.
5. Compute $R_i \bowtie S_i$ locally on all $H_i$.

\textsuperscript{23}A Bloom filter is a compact data structure that can be used to filter data according to a set of valid key values. We’ll discuss Bloom filters later in this course.