Data Processing on Modern Hardware

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

Summer 2014
Part V

Execution on Multiple Cores
Example: Star Joins

Task: run parallel instances of the query (↗ introduction)

```
SELECT SUM(lo_revenue)
FROM part, lineorder
WHERE p_partkey = lo_partkey
AND p_category <= 5
```

to implement  use either

- a hash join or
- an index nested loops join.
Execution on “Independent” CPU Cores

Co-run independent instances on different CPU cores.

Concurrent queries may seriously affect each other’s performance.
In Intel Core 2 Quad systems, two cores share an L2 Cache:

What we saw was cache pollution.

→ How can we avoid this cache pollution?
Dependence on cache sizes for some TPC-H queries:

Some queries are more sensitive to cache sizes than others.

- **cache sensitive**: hash joins
- **cache insensitive**: index nested loops joins; hash joins with very small or very large hash table
Locality Strength

This behavior is related to the **locality strength** of execution plans:

**Strong Locality**
- small data structure; reused very frequently
  - *e.g.*, small hash table

**Moderate Locality**
- frequently reused data structure; data structure $\approx$ cache size
  - *e.g.*, moderate-sized hash table

**Weak Locality**
- data not reused frequently or data structure $\gg$ cache size
  - *e.g.*, large hash table; index lookups
Execution Plan Characteristics

Locality effects how caches are used:

<table>
<thead>
<tr>
<th></th>
<th>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache pollution</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amount of cache used</td>
<td>small</td>
<td>large</td>
<td>large</td>
</tr>
<tr>
<td>amount of cache needed</td>
<td>small</td>
<td>large</td>
<td>small</td>
</tr>
</tbody>
</table>

Plans with **weak locality** have most severe impact on co-running queries.

Impact of co-runner on query:

<table>
<thead>
<tr>
<th></th>
<th>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong</td>
<td>low</td>
<td>moderate</td>
<td>high</td>
</tr>
<tr>
<td>moderate</td>
<td>moderate</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>weak</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>
4.2.1 Experiments

In order to understand locality strengths and related cache conflicts of hash join and index join, we ran experiments to test the performance degradation of hash join. The Y-axis is the performance degradation relative to the baseline cases. We made observations as follows.

1. When the hash table size is no larger than 1.1MB, two hash joins co-running suffer significant cache thrashing and miss penalty.
2. When the hash table size is between 1.1MB and 12.3MB, the performance degradations of hash joins caused by a co-running hash join are high.
3. When the hash table size is larger than 12.3MB, the hash join has weak locality.

Our experiments show that this setting performs pleasantly well in practice.

Table 1 summarizes performance degradations due to cache conflicts. There are mostly two kinds of cache conflict degradations: (1) cache pollution: an index join or a weak-locality hash join pollutes the LLC so that a strong-locality hash join co-runner with a hash table smaller than 392KB suffers high performance degradations. (2) Since cache pollution can affect its hash join co-runner with a hash table smaller than 392KB, we use a 2GB and a 4GB SSB data set. For the scale of 2GB, the size can further increase from 1.1MB to 12.3MB. When using index join to execute each query, the hash table size for a single expression is about 392KB. By the observations in paper [33] which shows that index joins are more than 20%. This is because, in practice, other components in the database may consume a small amount of cache space. (2) When the hash table size is larger than 392KB, the performance degradations of hash joins are similar to that of index joins. Our experiments provide us with a basis to distinguish locality strengths of the two operators. First, according to our experiment shows that their performance degradations are well in practice.

Figure 4: Performance degradations when co-running hash/index, hash/hash, and index/index.

Locality-Aware Scheduling

An optimizer could use knowledge about localities to schedule queries.

- **Estimate** locality during query analysis.
  - Index nested loops join → weak locality
  - Hash join:
    
    \[
    \text{hash table } \ll \text{cache size} \rightarrow \text{strong locality}
    \]
    
    \[
    \text{hash table } \approx \text{cache size} \rightarrow \text{moderate locality}
    \]
    
    \[
    \text{hash table } \gg \text{cache size} \rightarrow \text{weak locality}
    \]

- **Co-schedule** queries to minimize (the impact of) cache pollution.

> **Which queries should be co-scheduled, which ones not?**

- Only run weak-locality queries next to weak-locality queries.
  
  → They cause high pollution, but are not affected by pollution.

- Try to co-schedule queries with small hash tables.
Experiments: Locality-Aware Scheduling

PostgreSQL; 4 queries (different \texttt{p\_category}s); for each query: $2 \times$ hash join plan, $2 \times$ INLJ plan; impact reported for hash joins:

<table>
<thead>
<tr>
<th>Hash Table Size</th>
<th>Performance Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.78 MB</td>
<td>-10%</td>
</tr>
<tr>
<td>2.26 MB</td>
<td>-20%</td>
</tr>
<tr>
<td>4.10 MB</td>
<td>-30%</td>
</tr>
<tr>
<td>8.92 MB</td>
<td>-40%</td>
</tr>
</tbody>
</table>

Source: Lee et al. VLDB 2009.
Weak-locality plans cause cache pollution, because they use much cache space even though they do not strictly need it.

By partitioning the cache we could reduce pollution with little impact on the weak-locality plan.

But:
- Cache allocation controlled by hardware.
Remember how caches are organized:

- The **physical address** of a memory block determines the **cache set** into which it could be loaded.

Thus,

- We can **influence hardware behavior** by the **choice of physical memory allocation**.
Page Coloring

The address ↔ cache set relationship inspired the idea of page colors. Each memory page is assigned a color. Pages that map to the same cache sets get the same color.

How many colors are there in a typical system?

---

5 Memory is organized in pages. A typical page size is 4 kB.
By using memory only of certain colors, we can effectively restrict the cache region that a query plan uses.

Note that

- Applications (usually) have no control over physical memory.
- Memory allocation and virtual ↔ physical mapping are handled by the operating system.
- We need OS support to achieve our desired cache partitioning.
MCC-DB ("Minimizing Cache Conflicts"):  
- Modified Linux 2.6.20 kernel  
  - Support for **32 page colors** (4 MB L2 Cache: 128 kB per color)  
  - **Color specification** file for each process (may be modified by application at any time)  

- Modified instance of PostgreSQL  
  - **Four colors** for regular buffer pool  
  
  - **Implications on buffer pool size** (16 GB main memory)?  

- For **strong- and moderate-locality** queries, allocate colors as needed (i.e., as estimated by query optimizer)
Experiments

Moderate-locality hash join and weak-locality co-runner (INLJ):

![Graph with L2 Cache Miss Rate vs Colors to Weak- Locality Plan]

- Weak locality (INLJ)
- Moderate locality (HJ)
- Single-threaded execution

Source: Lee et al. VLDB 2009.
Experiments

Moderate-locality hash join and weak-locality co-runner (INLJ):

![Graph showing execution time vs colors to weak-locality plan]

- Weak locality (INLJ)
- Single-threaded execution
- Moderate locality (HJ)
- Single-threaded execution

Source: Lee et al. VLDB 2009.
Experiments: MCC-DB

PostgreSQL; 4 queries (different `p_category`); for each query: $2 \times$ hash join plan, $2 \times$ INLJ plan; impact reported for hash joins:

**hash table size**

- 0.78 MB
- 2.26 MB
- 4.10 MB
- 8.92 MB

**performance impact**

- Default
- Locality-aware
- Page coloring

Source: Lee et al. VLDB 2009.
Databases are often faced with **highly concurrent workloads**.

**Good news:**
- Exploit parallelism offered by hardware (increasing number of cores).

**Bad news:**
- Increases relevance of **synchronization mechanisms**.

Two levels of synchronization in databases:

**Synchronize on User Data**
- to guarantee transaction semantics; database terminology: **locks**

**Synchronize on Database-Internal Data Structures**
- short-duration locks; called **latches** in databases

We’ll now look at the latter, even when we say “locks.”
There are two strategies to implement locking:

**Blocking** (operating system service)
- **De-schedule** waiting thread until lock becomes free.
- Cost: two context switches (one to sleep, one to wake up)
  - $\rightarrow \approx 12–20 \mu s e c$

**Spinning** (can be done in user space)
- Waiting thread repeatedly polls lock until it becomes free.
- Cost: two cache miss penalties (if implemented well)
  - $\rightarrow \approx 150 n s e c$
- Thread burns CPU cycles while spinning.
Implementation of a spinlock?
Thread Synchronization

**Blocking:**

- thread working
- lock held
- thread 1
- de-schedule
- thread 2
- wake-up

**Spinning:**

- thread working
- lock held
- thread 1
- thread spinning
- short delay
- thread 2
Experiments: Locking Performance

Sun Niagara II (64 hardware contexts):

Throughput (ktps)

Blocking

Spinning

Ideal

100% load

# Threads

0 32 64 96 128 160 192

Source: Johnson et al. Decoupling Contention Management from Scheduling. ASPLOS 2010.
Under **high load**, spinning can cause problems:

- More threads than hardware contexts.
- Operating system **preempts** running task.
  - Working and spinning threads all appear busy to the OS.
  - Working thread likely had longest time share already → gets **de-scheduled** by OS.
- **Long** delay before working thread gets re-scheduled.
- By the time working thread gets re-scheduled (and can now make progress), waiting thread likely gets de-scheduled, too.
In contrast to blocking, spinning or "busy waiting" schemes (grouped on the left side of Figure 2) leave waiting threads on the critical path. However, the same FIFO ordering makes such algorithms especially vulnerable to preemptions because they form a FIFO queue and each lock handoff targets a specific thread ("MCS" in Figure 2). Queue-based locks also give each thread its own memory location to spin on, eliminating unnecessary coher- tion drops rapidly when the OS scheduler preempts lock holders. Further, the orderly handoff is an elegant solution for "thundering herd" problem, where all waiting threads race for CPU time is wasted spinning on contended locks, and that frac- tion is less than 100% and contention is low. However, as soon as utilization passes 100% priority inversions quickly dominate, as doing useful work, spinning due to contention, and spinning due to memory interference with computation. Finally, the OS scheduler cannot implement the design that minimizes the need for context switching under low traffic. However, the same FIFO ordering makes such algorithms undesirable side effects on load (see below). Heavyweight OS implementations create heavy traffic in the memory system and threads might have been able to use. In addition, naive spinlock on the critical path. However, it also wastes CPU time other delays per handoff) and avoids context switching or system calls thus interfere with computation. Finally, the OS scheduler cannot cooperate with the OS scheduler, and are the focus of this work.

Where spin-then-yield schemes are essentially spinlocks which use capability to both sleep and wake threads allows threads to block without timeouts, without the risk of leaving a contended lock idle. Where spin-then-block schemes are essentially spinlocks which use cooper- ation with the OS scheduler as a form of backoff, hybrid spin-then-block schemes instead of passing the lock to them. By only handing the lock to lock holders to remove preempted threads from the lock queue. Time-published locks only protect the queue, leaving lock holders vul- nerable to preemptions because time-published MCS locks ("TP-MCS" in Figure 2) allow cooperation with the OS scheduler. Preempted lock holders impact all locks which do not publish locks to the OS scheduler, and other threads cannot bypass it even if it was preempted again, and other threads cannot bypass it even if it was preempted, only to have the new thread waste its time slice spinning. Hence, we run a database telecommunication benchmark (TM-1) on a 64-context machine (see Section 4 for details), using a state-of-the-art spinlock. We instrument the code to differentiate between spin- the scheduler as a form of backoff, hybrid spin-then-block schemes instead of passing the lock to them. By only handing the lock to lock holders to remove preempted threads from the lock queue. Time-published locks only protect the queue, leaving lock holders vulnerable to preemptions because time-published MCS locks ("TP-MCS" in Figure 2) allow cooperation with the OS scheduler. Preempted lock holders impact all locks which do not publish locks to the OS scheduler, and other threads cannot bypass it even if it was preempted, only to have the new thread waste its time slice spinning. Hence, we run a database telecommunication benchmark (TM-1) on a 64-context machine (see Section 4 for details), using a state-of-the-art spinlock. We instrument the code to differentiate between spin- the scheduler as a form of backoff, hybrid spin-then-block schemes instead of passing the lock to them. By only handing the lock to lock holders to remove preempted threads from the lock queue. Time-published locks only protect the queue, leaving lock holders vulnerable to preemptions because time-published MCS locks ("TP-MCS" in Figure 2) allow cooperation with the OS scheduler. Preempted lock holders impact all locks which do not publish locks to the OS scheduler, and other threads cannot bypass it even if it was preempted, only to have the new thread waste its time slice spinning. Hence, we run a database telecommunication benchmark (TM-1) on a 64-context machine (see Section 4 for details), using a state-of-the-art spinlock. We instrument the code to differentiate between spin- the scheduler as a form of backoff, hybrid spin-then-block schemes instead of passing the lock to them. By only handing the lock to lock holders to remove preempted threads from the lock queue. Time-published locks only protect the queue, leaving lock holders vulnerable to preemptions because time-published MCS locks ("TP-MCS" in Figure 2) allow cooperation with the OS scheduler. Preempted lock holders impact all locks which do not publish locks to the OS scheduler, and other threads cannot bypass it even if it was preempted, only to have the new thread waste its time slice spinning. Hence, we run a database telecommunication benchmark (TM-1) on a 64-context machine (see Section 4 for details), using a state-of-the-art spinlock. We instrument the code to differentiate between spin- the scheduler as a form of backoff, hybrid spin-then-block schemes instead of passing the lock to them. By only handing the lock to lock holders to remove preempted threads from the lock queue. Time-published locks only protect the queue, leaving lock holders vulnerable to preemptions because time-published MCS locks ("TP-MCS" in Figure 2) allow cooperation with the OS scheduler. Preempted lock holders impact all locks which do not publish locks to the OS scheduler, and other threads cannot bypass it even if it was preempted, only to have the new thread waste its time slice spinning. Hence, we run a database telecommunication benchmark (TM-1) on a 64-context machine (see Section 4 for details), using a state-of-the-art spinlock. We instrument the code to differentiate between spin-
The properties of spinning and blocking suggest their use for different purposes:

- **Spinning** features quick lock hand-offs.
  → Use spinning to coordinate access to a shared data structure (contention).

- **Blocking** reduces system load (→ scheduling).
  → Use blocking at longer time scales.
  → Block when system load increases to reduce scheduling overhead.

**Idea:** Monitor system load (using a separate thread) and control spinning/blocking behavior off the critical code path.
The **load controller** periodically

- Determines current load situation from the OS.
- If system gets **overloaded**
  - “invite” threads to block with help of a **sleep slot buffer**.
  - Size of sleep slot buffer: number of threads that should block.
- When load gets less
  - controller **wakes up** sleeping threads, which register in sleep slot buffer before going to sleep.
A thread that wants to acquire a lock

- Checks the regular spin lock.
- If the lock is already taken, it tries to enter the sleep slot buffer and blocks (otherwise it spins).
- The load controller will wake up the thread in time.
Controller Overhead

The graph illustrates the impact of varying contention for 95% and 150% load, with the x-axis representing update delay (μs) and the y-axis throughput (ktps). It shows a comparison of 98%, 110%, and 150% load conditions.

- **98% load** demonstrates consistent throughput irrespective of update delay.
- **110% load** shows a slight decrease in throughput as update delay increases, indicating slight contention effects.
- **150% load** experiences a pronounced drop in throughput with more significant delays, highlighting increasing contention levels.

The graph supports the observation that load control is effective in managing system throughput under varying load conditions, maintaining performance even as contention increases.
Performance Under Load

Source: Johnson et al. Decoupling Contention Management from Scheduling. *ASPLOS 2010.*