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?
Cache Sensitivity

Dependence on cache sizes for some TPC-H queries:

(a) L2 Miss Rate
(b) CPI

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>cache pollution</th>
<th>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<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>strong</th>
<th>moderate</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>strong</td>
<td>low</td>
<td>moderate</td>
</tr>
<tr>
<td>moderate</td>
<td>moderate</td>
<td>high</td>
</tr>
<tr>
<td>weak</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 use SSB-based synthetic queries to characterize them. These queries involve temporary data in the database may consume a small amount of cache space. (2) cache pollution: an index join or a weak-locality hash join pollutes the LLC so that a strong-locality hash join can affect its hash join co-runner with a hash table smaller than 9.28MB. For the scale of 4GB, the size can further increase to 18.6MB. When using index join to execute each query, the performance degradation relative to the baseline case. Then we run two queries using the aforementioned tables.

In this experiment, we examine two combinations for the queries: (1) co-running two index joins for the queries, in the ascending order. For brevity, we said three combinations to measure the performance degradation. The Y-axis is the performance degradation relative to the baseline cases. Figure 4 shows the results of the experiment as the baseline case. Then we run two queries using the aforementioned tables.

The X-axis values are the hash table sizes of hash joins, relative to the baseline cases. Figure 4 shows the results of the experiment as the baseline case. Then we run two queries using the aforementioned tables.

Table 1 summarizes performance degradations due to cache conflicts. There are mostly two kinds of cache conflict degradations: (1) performance degradation: (1) when the hash table sizes are larger than 12.3MB, 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.

We select these queries because a very common pattern in Multi-core Processors for Databases. VLDB 2009. We examine three combinations to measure the performance degradation: (1) co-running two hash joins (hash/hash), (2) co-running two index joins (hash/index). For hash/hash and index/index, we have a sum function on the LOCATIONS and the LINEORDER table. In this experiment, we examine two synthetic queries to characterize them. These queries involve temporary data in the database may consume a small amount of cache space. (2) cache pollution: an index join or a weak-locality hash join pollutes the LLC so that a strong-locality hash join can affect its hash join co-runner with a hash table smaller than 9.28MB. For the scale of 4GB, the size can further increase to 18.6MB. When using index join to execute each query, the performance degradation relative to the baseline case. Then we run two queries using the aforementioned tables.

In order to understand locality strengths and related cache conflicts of hash join and index join, we use SSB-based synthetic queries to characterize them. These queries involve temporary data in the database may consume a small amount of cache space. (2) cache pollution: an index join or a weak-locality hash join pollutes the LLC so that a strong-locality hash join can affect its hash join co-runner with a hash table smaller than 9.28MB. For the scale of 4GB, the size can further increase to 18.6MB. When using index join to execute each query, the performance degradation relative to the baseline case. Then we run two queries using the aforementioned tables.

In this experiment, we examine two combinations for the queries: (1) co-running two index joins for the queries, in the ascending order. For brevity, we said three combinations to measure the performance degradation. The Y-axis is the performance degradation relative to the baseline cases. Figure 4 shows the results of the experiment as the baseline case. Then we run two queries using the aforementioned tables.

The X-axis values are the hash table sizes of hash joins, relative to the baseline cases. Figure 4 shows the results of the experiment as the baseline case. Then we run two queries using the aforementioned tables.

Table 1 summarizes performance degradations due to cache conflicts. There are mostly two kinds of cache conflict degradations: (1) performance degradation: (1) when the hash table sizes are larger than 12.3MB, 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.

We select these queries because a very common pattern in Multi-core Processors for Databases. VLDB 2009. We examine three combinations to measure the performance degradation: (1) co-running two hash joins (hash/hash), (2) co-running two index joins (hash/index). For hash/hash and index/index, we have a sum function on the LOCATIONS and the LINEORDER table. In this experiment, we examine two synthetic queries to characterize them. These queries involve temporary data in the database may consume a small amount of cache space. (2) cache pollution: an index join or a weak-locality hash join pollutes the LLC so that a strong-locality hash join can affect its hash join co-runner with a hash table smaller than 9.28MB. For the scale of 4GB, the size can further increase to 18.6MB. When using index join to execute each query, the performance degradation relative to the baseline case. Then we run two queries using the aforementioned tables.

In order to understand locality strengths and related cache conflicts of hash join and index join, we use SSB-based synthetic queries to characterize them. These queries involve temporary data in the database may consume a small amount of cache space. (2) cache pollution: an index join or a weak-locality hash join pollutes the LLC so that a strong-locality hash join can affect its hash join co-runner with a hash table smaller than 9.28MB. For the scale of 4GB, the size can further increase to 18.6MB. When using index join to execute each query, the performance degradation relative to the baseline case. Then we run two queries using the aforementioned tables.
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:

![Graph showing performance impact vs. hash table size](chart)

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.

  ┌───────────────┐  ┌────────────────────┐
  │       ────────│  │                   │
  │     ──────────┤  │          ──────┘
  │  ────────────│  │           ──────
  │             └─────┬────────────┘
  └───────────────┘  └────────────────────┘

  ┌───┬───────────┐
  │   │          │
  │ tag│set index │offset
  └───┴───────────┘

  ┌───┬───────────┐
  │   │          │
  │ byte address     │ block address
  └───┴───────────┘

Thus,

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

---

7Memory 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 showing L2 Cache Miss Rate vs. Colors to Weak-Locality Plan]

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

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

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

![Performance impact graph]

Source: Lee et al. VLDB 2009.

© Jens Teubner · Data Processing on Modern Hardware · Summer 2013
Highly Concurrent Workloads

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)
  \[ \rightarrow \approx 12\text{–}20 \mu\text{sec} \]

**Spinning** (can be done in user space)
- Waiting thread repeatedly *polls* lock until it becomes free.
- Cost: two *cache miss penalties* (if implemented well)
  \[ \rightarrow \approx 150 \text{ nsec} \]
- Thread burns CPU cycles while spinning.
Implementation of Spinlocks

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)

0 30 60 90 120 150

0 32 64 96 128 160 192

# Threads

Blocking

Spinning

Ideal

Source: Johnson et al. Decoupling Contention Management from Scheduling. ASPLOS 2010.
Spinning Under High Load

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.
Spinning

In contrast to blocking, spinning or "busy waiting" schemes (grouped on the left side of Figure 2) leave waiting threads on the critical path. Leading to situations where a lock holder gets preempted, only to have the new thread waste its time slice spinning. Thus, interfere with computation. Finally, the OS scheduler cannot implement the capability to both sleep and wake threads allows threads to block on the critical path. However, it also wastes CPU time other delays per handoff) and avoids context switching or system calls.

2.1 Preemption-resistant Spinlocks

Preempted lock holders impact all locks which do not being the lock's poor performance we modify the TM-1 benchmark to show how severe the problem of preempted lock holders can be, 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-then-yield and pure spinning.

2.2 Backoff and Spin-then-block Hybrids

Backoff-based spinning provides another solution to the "thundering herd" problem by limiting the number of waiting threads which can respond simultaneously. Test-and-set with exponential backoff [1] and spin-then-yield variants [6][27], fall into this category, with the latter removing 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

Machine Util (%)

<table>
<thead>
<tr>
<th>Client Threads</th>
<th>Work</th>
<th>Contention</th>
<th>Prio-Invert</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>31</td>
<td>15</td>
<td>44</td>
</tr>
<tr>
<td>31</td>
<td>47</td>
<td>31</td>
<td>24</td>
</tr>
<tr>
<td>47</td>
<td>63</td>
<td>47</td>
<td>26</td>
</tr>
<tr>
<td>63</td>
<td>71</td>
<td>63</td>
<td>28</td>
</tr>
<tr>
<td>71</td>
<td>95</td>
<td>71</td>
<td>28</td>
</tr>
<tr>
<td>95</td>
<td>127</td>
<td>95</td>
<td>28</td>
</tr>
<tr>
<td>127</td>
<td>159</td>
<td>127</td>
<td>28</td>
</tr>
<tr>
<td>159</td>
<td>191</td>
<td>159</td>
<td>28</td>
</tr>
</tbody>
</table>

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

Throughput (ktps)

98% load
110% load
150% load

Update delay (µs)

Effect of changing the load controller update interval

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

ASPLOS 2010.

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