

# Cache-Aware Database Systems Internals

Chapter 7



1

## Data Placement in RDBMSs

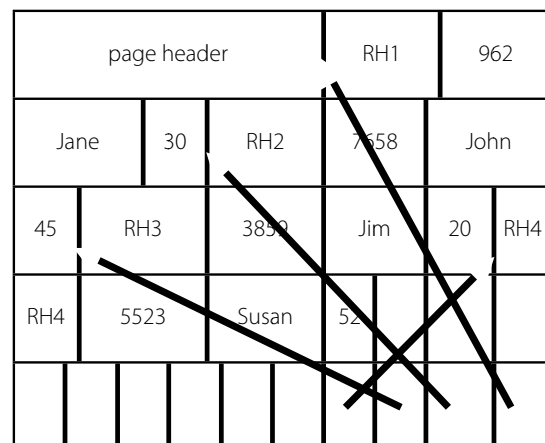
- A careful analysis of query processing operators and data placement schemes in RDBMS reveals a paradox:
    - Workloads perform sequential traversals through parts of the relation that are *not* referenced.
1. Requested items (attributes) and transfer unit between memory and processor mismatch in size,
  2. data placement schemes store data in different order than they are accessed.

# The “Whole Tuple” Model

- Query operators typically access only a small fraction of a whole tuple (record).
  1. Loading the cache with useless data wastes bandwidth.
  2. Cache is polluted with useless data.
  3. Useless data may enforce the replacement of valuable data that is actually needed by the query processor.

# N-Ary Storage Model (NSM)

RID	SSN	Name	Age
1	962	Jane	30
2	7658	John	45
3	3859	Jim	20
4	5523	Susa	52
5	9743	Leon	43
6	618	Dan	47



- Record header (RH) contains **NULL** bitmap, offset to variable length attributes, *etc.*

# NSM and Cache Pollution

- Depending on the query, NSM may exhibit pool cache performance:

```
SELECT  Name
FROM    R
WHERE   Age < 40
```

- To evaluate the predicate, employ a scan operator to retrieve the **Age** attribute values.
- If cache block size < record size, the scan operator will incur one cache miss per record.

# Decomposed Storage Model

- The Decomposed Storage Model (DSM) is the other extreme. Relations undergo full vertical partitioning.

- $n$ -ary relation  $\Rightarrow n$  binary subrelations (**RID**, *attr*).
- Sequential accesses (scan) lead to high spatial locality if queries touch few attributes.
- But: Multi-attribute queries call for subrelation joins.

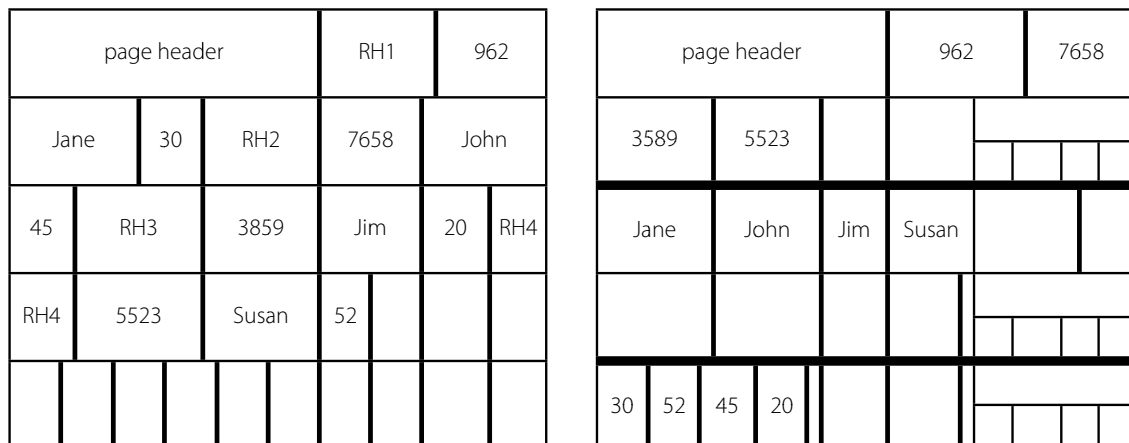
RID	Age
1	30
2	45
3	20
4	52
5	43
6	47

(... ORDER BY Age, Name ?)

# PAX Data Placement

- Partition Attributes Across (PAX):
  - Try to maximize inter-record spatial locality within each column and within each page.
  - Try to minimize record construction cost.
  - Low impact on the rest of the DBMS kernel.
- PAX:
  - Cache-friendly placement of attributes *inside each page* (“DSM inside an NSM page”).

# PAX Data Page Layout



- Data remains on its original NSM page. NSM and PAX storage requirements on par.
- Attribute values are grouped into minipages.

# NSM vs. DSM vs. PAX

	NSM	DSM	PAX
Inter-record spatial locality	✗	✓	✓
Low record reconstruction cost	✓	✗	✓

- Local changes to the page manipulation code of an NSM-style DBMS kernel.
- Note: NSM may still win for transaction-processing-style workloads (“wide queries”), DSM may still win for complex decision support queries (“narrow”).

# DBMS Instruction Footprints

- In DBMS, memory stall times are dominated by L2 data cache misses and L1 instruction cache misses.
- Typical query workloads exhibit active instruction footprints which clearly exceed L1 i-cache size.
- Volcano model: operator returns control to parent operator immediately after generating one tuple.

Instructions for this operator get evicted from L1 i-cache and need to be reloaded for next tuple  
⇒ Instruction cache thrashing.

# Branch History Thrashing

- The classical query execution model also incurs another of i-cache pollution: branch mispredictions.
- Operators check for nullability, data types, comparison outcome, over/underflows, error conditions, ... per record.
- CPU keeps track of branch histories (typically 512–4096 history slots).
- Large instructions footprints with many branch instructions will thrash the branch history unit.

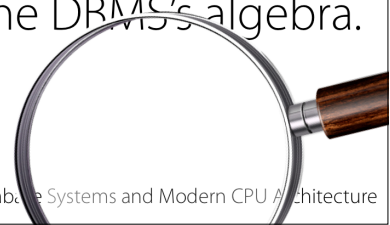
# Operator Instruction Footprints (PostgreSQL)

Instruction Footprints

TableScan	without predicates	9K
	with predicates	13K
IndexScan		14K
Sort		14K
Nested Loop Join	(inner) TableScan	11K
	(inner) IndexScan	11K
MergeJoin		12K
HashJoin	build	10K
	probe	12K
Aggregation	<b>COUNT</b>	< 1K
	<b>MAX, MIN</b>	< 1.6K

# Instruction Footprint Analysis

- Naive: Study DBMS code to construct static call graph of functions reachable from an operator.
  - Sums up sizes of object code of all *possibly* called functions  $\Rightarrow$  significant overestimation of size.
- Assume that dynamic function call behavior is largely data independent:  
Calibrate core DBMS by performing a set of queries over “mini” data instances covering the DBMS’s algebra.
  - Shared code only counts once.



# Buffer Operators

- Given a parent–child pair ( $p$ – $c$ ) of operators in a query plan, introduce a new buffer operator in between.
  - 1. During query execution, the buffer operator stores a large array of pointers to tuples generated by  $c$ .
  - 2. Buffer operators fills its pointer array, control is *not* returned to  $p$  until buffer is fully filled.
  - 3. Once buffer filled (or  $c$  exhausted), return control to  $p$  and serve tuples from the buffer.

$p$

|

|

$c$

# Execution Patterns

- Query operator interleaving without buffer operator:

*pcpcpcpcpcpcpcpcpcpcpcpcpcpcpc*

- With buffer operator (buffer size: 5 tuple pointers):

*pccccpppppccccpppppccccppppp*

- Avoid i-cache thrashing if the instruction size of  $p$  plus  $c$  exceeds i-cache size. At least 4 out of 5 executions of operators will find their instructions cached.

# Placing Buffer Operators

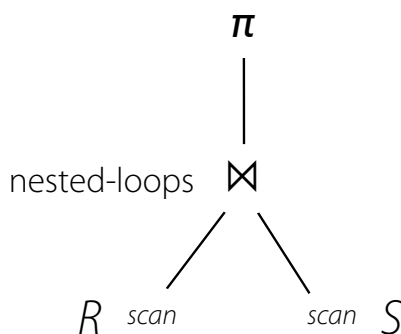
- We aim to improve spatial instruction locality. Place a buffer operator above a group of operators whose effective instruction footprint fits into the L1 i-cache:
  1. Consider each plan leaf operator to be an execution group of its own.
  2. Traverse query tree bottom-up, add operator to current group if adding its footprint does not grow group beyond L1 i-cache size.

# Buffering “Too Much”

- Buffer operators introduce overhead during query execution. Do *not* introduce buffer operator for operator group, if
  - expected cardinality of tuple flow in group is small: operators are only executed a few times (“calibrate” DBMS: measure plan performance with/without buffering at various cardinalities),  
or
  - operator group contains a blocking operator (sort, hash-table building).

# Rescheduling Operators

- Do not alter given plan but try to schedule identical operators together (share object code):



- Scheduling the two table scans *scan* together will save i-cache capacity.
- Relations  $R, S$  need to be materialized in memory, however.

# Hash Join

- Hash join is one (of many possible) physical implementations of the equi-join operator of the relational algebra.
- Hash join is a viable choice to implement join, whenever
  - there is no exploitable order on the input relations, or
  - the input relations are intermediate results with no index support  $\Rightarrow$  try to construct index on-the-fly.

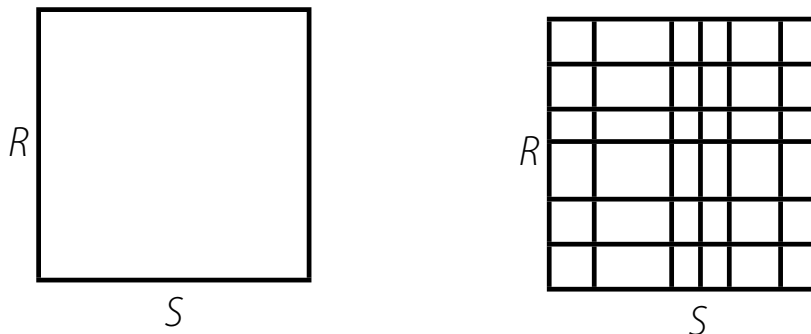
# Building Temporary Indexes

- The construction of a temporary index for a single join may pay off if the input cardinalities are high.
- In the case of hash join, build a hash table on the join attribute of the smaller input relation (*build input*).
  - Make sure that hash table fits into main memory:

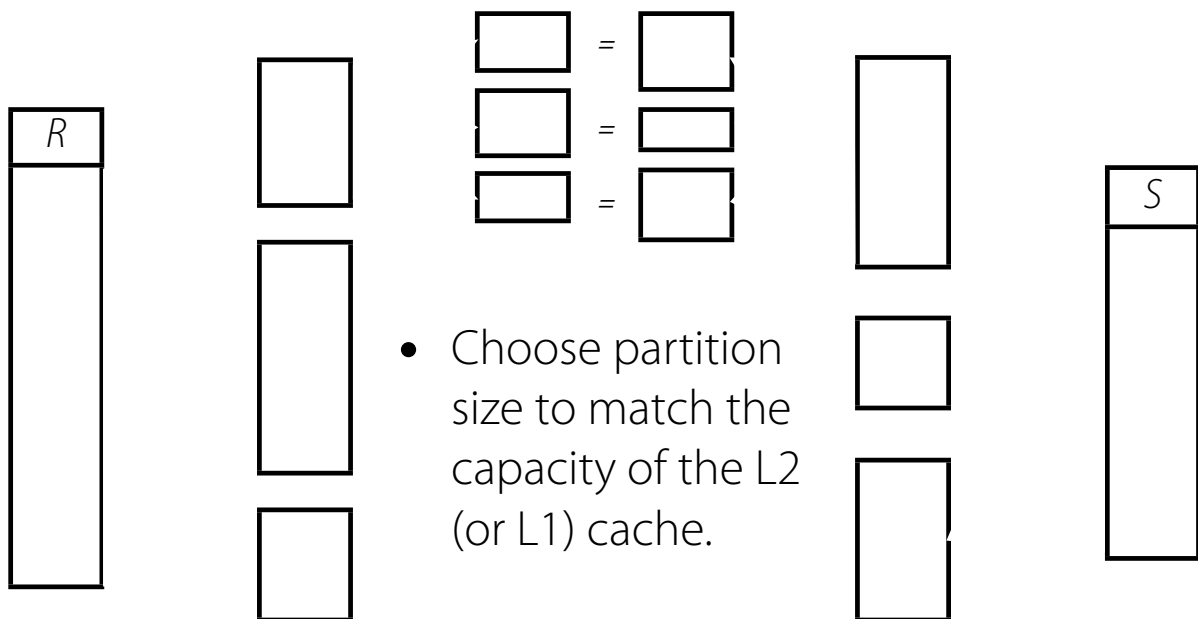
Due to the nature of hashing (*no locality*), almost all lookups in the hash table will lead to page misses.

# Partitioned Hash Join

- Partition both input relations. Potentially matching tuples will reside in corresponding partitions.
- ⇒ Resulting partitions fit into memory: hash table for partitions will fit into memory and not incur page misses.

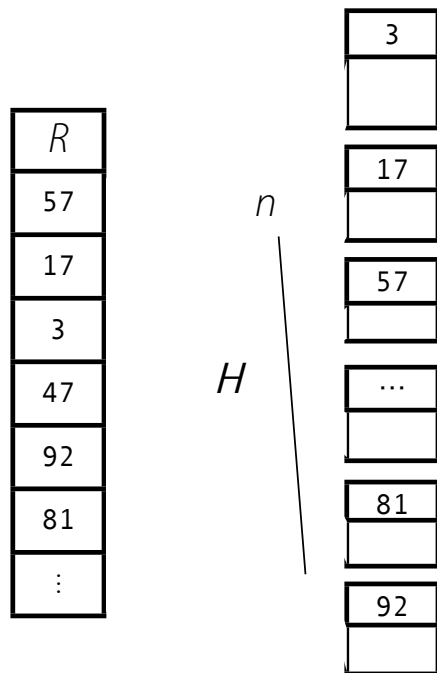


# In-Memory Partitioned Hash Join



- Choose partition size to match the capacity of the L2 (or L1) cache.

# One-Pass Hash Clustering



- Apply hash function  $H$  to distribute tuples into their partitions.
  - Partition size chosen to fit into L1 cache  $\Rightarrow$  large  $n$ .
  - Random access pattern that writes into  $n$  locations.
- $\Rightarrow$  Cache and TLB thrashing.

# Controlling the Fan-Out

- Problem: Need to split into  $H$  partitions to fit into L1 cache blocks size — but that many data streams will trash the TLB or exceed the number of cache blocks.
- Idea: Radix Clustering:  
Split the relation in multiple, say  $P$ , passes.
  - In each pass  $p$  ( $1 \leq p \leq P$ ) only consider only  $B_p$  bits of the join column to control the fan-out.
  - That pass will write to  $H_p = 2^{B_p}$  partitions. Choose  $B_p$  so as to not thrash the TLB/cache.
  - Choose  $P$  and considered bits such that

$$\prod_{p \in 1 \dots P} H_p \geq H.$$

# Two-Pass Radix Clustering

$R$	
57	(001)
17	(001)
3	(011)
47	(111)
92	(100)
81	(001)
20	(100)
6	(110)
96	(000)
37	(101)
66	(010)
75	(001)

$H_1$

	$H_1(R)$	
00	57	(001)
	17	(001)
	81	(001)
	96	(000)
	75	(001)
01	3	(011)
	66	(010)
10	92	(100)
	20	(100)
	37	(101)
11	47	(111)
	6	(110)

$H_2$

	$H_2(H_1(R))$	
0	96	(000)
1	57	(001)
	17	(001)
	81	(001)
	75	(001)
0	66	(010)
1	3	(011)
0	92	(100)
	20	(100)
1	37	(101)
0	6	(110)
1	47	(111)

# Joining Radix-Clustered Relations

- After relation  $R$  has been radix-clustered on  $B = \sum_{p \in 1 \dots P} B_p$  bits, each partition will typically contain  $|R| / 2^B$  tuples.
- Use regular (nested-loops) join algorithm to join tuples in associated partitions.

	$R$
(000)	96
(001)	57
(001)	17
(001)	81
(001)	75
(010)	66
(011)	3
(100)	92
(100)	20
(101)	37
(110)	6
(111)	47

$S$	
96	(000)
32	(000)
17	(001)
10	(010)
2	(010)
66	(010)
3	(011)
35	(011)
20	(100)
47	(111)

# Joins and Projections in DSM

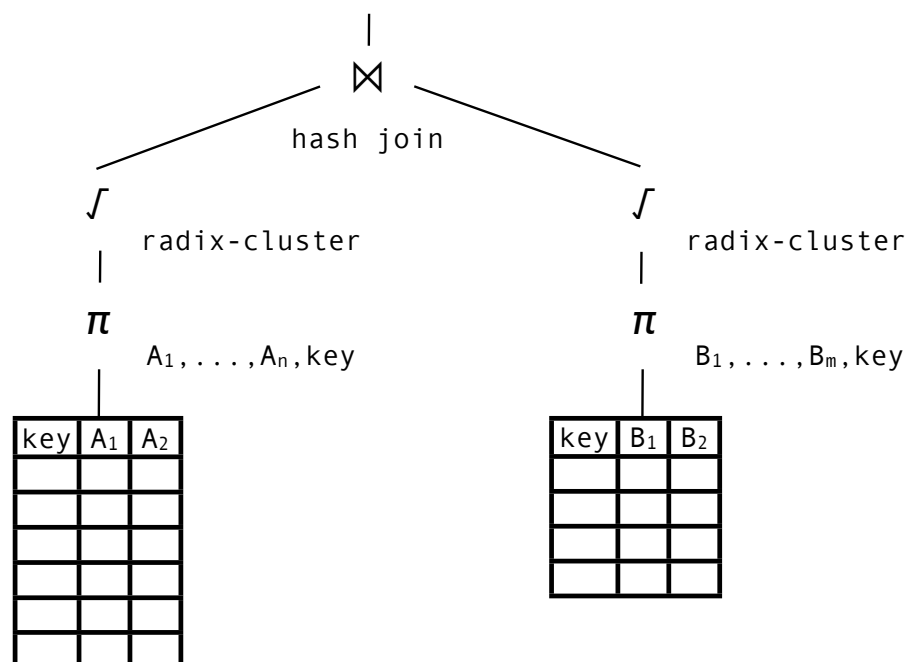
- Like any other relation, in DSM a join result consists of two columns — regardless of the schema of the query containing the join.

– Consider:

```
SELECT  R.A1, ..., R.An,  
        S.B1, ..., S.Bm  
FROM    R, S  
WHERE   R.key = S.key
```

- How will the projection (**SELECT** clause) be handled?

# Pre-Projection (NSM)



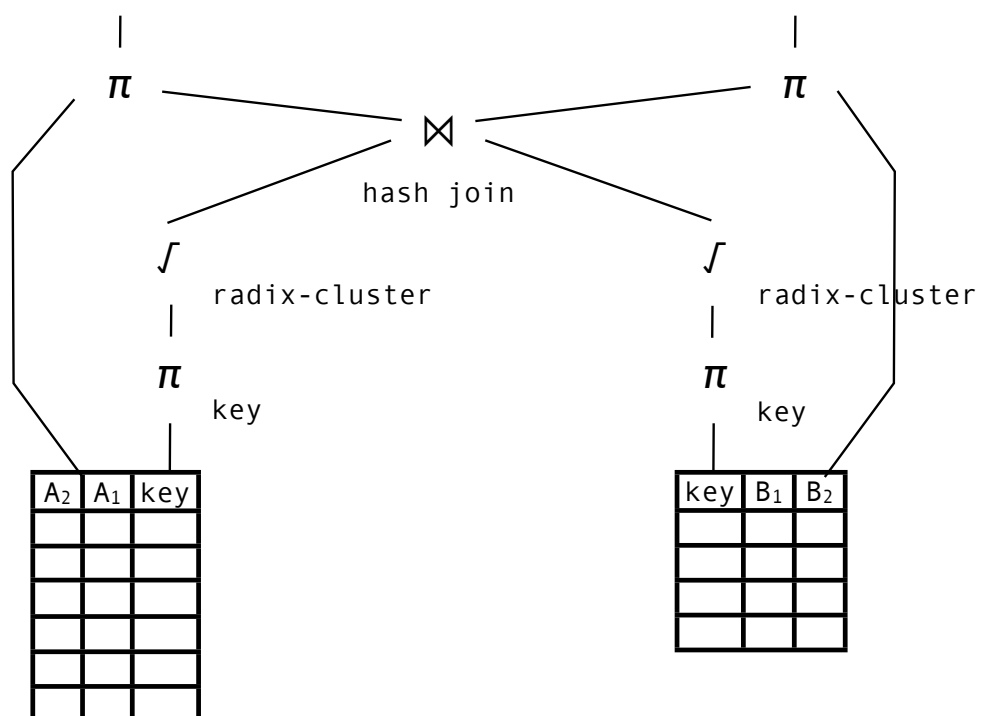
# Joins and Projections in DSM

- DSM join result contains pairs of row identifiers (RIDs) of qualifying tuples (a so-called join index).
- To construct the fully projected join result, perform RID-based lookups in the join input relations.

$R$					
$A_1$		$A_2$		$A_3$	
0	A	0	E	0	x
1	A	1	E	1	z
2	B	2	F	2	z
3	D	3	H	3	y
4	C	4	G	4	z
5	D	5	H	5	z
6	D	6	H	6	y
7	D	7	H	7	y

$R \bowtie S$	
3	3
3	5
4	4
0	2
7	0
7	1

# Post-Projection



# Post-Projection and Partitioned Hash Join

- But: Due to the nature of partitioned hash join, the RIDs (MonetDB: `oid` values) of *neither input relation* will appear sorted in the two-column join index.  
Standard improvement:
  1. Sort join index in order of RIDs of larger relation.
  2. Access tuples of larger relation using a sequential scan (spatial locality).
  3. Tuples of smaller relation accessed randomly but may fit into the cache (temporal locality).

# Projection Using Radix Cluster

- Fully sorting the join index is overkill. A partial ordering can also achieve the same effect:
  - Cluster the join index, each cluster contains RIDs of a disjoint range.
  - Choose cluster size such that the contained RIDs point to a limited region in the projection column.
  - If this region fits into the cache, post-projection will approach optimal cache (re-)usage.

# Partial Radix Cluster

$R$	$\bowtie$	$S$
3		3
3		5
4		4
0		2
7		0
7		1

0	3
1	3
2	0
3	4
4	7
5	7

$R$					
$A_1$		$A_2$		$A_3$	
4	C	4	G	4	z
5	D	5	H	5	z
6	D	6	H	6	y
7	D	7	H	7	y

(0??)	3	3
(0??)	3	5
(0??)	0	2
(1??)	4	4
(1??)	7	0
(1??)	7	1

Partial Radix Cluster (1 bit)

MonetDB/MIL:  
`mark()`

$\pi_R(R \bowtie S)$					
$A_1$		$A_2$		$A_3$	
0	D	0	H	0	y
1	D	1	H	1	y
2	A	2	E	2	x
3	C	3	G	3	z
4	D	4	H	4	y
5	D	5	H	5	y

## Database Operation and Simultaneous Multithreading

- Simultaneous multithreading (SMT) improves CPU performance by supporting thread-level parallelism on a single processor.
  - An SMT processor pretends to be multiple logical processors.
  - Multiple threads may issue instructions on each cycle and share processor resources.
- SMT on Intel® CPUs (Xeon, Pentium 4):  
*"Hyper-Threading Technology"*

# Simultaneous Multithreading

- In an SMT systems, threads share the memory bus and the caches.
- SMT intrinsically performs worse than a system with multiple physical CPUs (threads may even compete for a shared resource).
- Improvement of single-threaded CPUs: Higher instruction throughput (*e.g.*, continue to run thread *A* while thread *B* waits for cache miss).
- Idea: Use shared cache to communicate between threads.

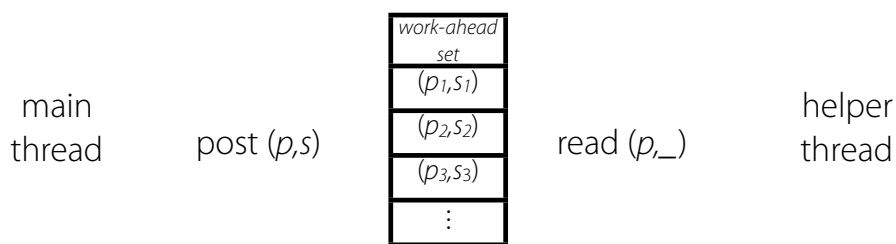
# Bi-Threaded Operators

- Try to perform thread-based data cache preloading:
  1. Operator execution itself happens in a main thread.
  2. A separate helper thread works ahead of the main thread and preloads data that will soon be needed by the main thread.
- Data preloading in helper thread can be overlapped with CPU computation in main thread.
- Helper thread suffers most of the memory latency, main thread will experience a higher cache hit rate.

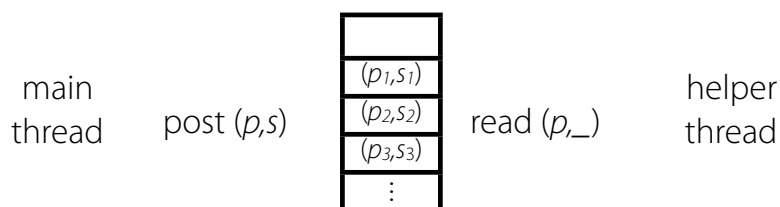


# Work-Ahead Set

1. Main thread posts soon-to-be-needed memory references  $p$  (plus a small amount of state information  $s$ ) into work-ahead set:  $\{(p,s)\}$ .
2. Helper thread reads  $(p, \_)$  from work-ahead set, accesses  $p$  and thus populates the data cache.
  - Note: Correct operation of main thread is *not* dependent on helper thread.



# Work-Ahead Set



Cache

RAM

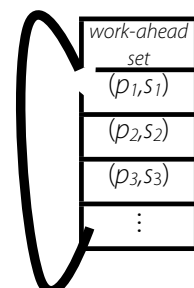
- Why not use prefetching in the main thread?
  - Requires manual computation of prefetching distance and scheduling of prefetch instructions.
  - Prefetches are *not* binding and may be dropped by the CPU (e.g., if a TLB miss would result).

# Staged Data Access

- Main operator threads need to be written to access data in stages.
  - Example: In hash join, separate probe of hash table and construction of output tuple (post-projection). Two stages for a probe/construction:
    1. Main thread posts address  $p$  of probed hash bucket to work-ahead set. Then continues with *other computation* (e.g., hashing of further join keys).
    2. Later, main thread removes  $p$  from work-ahead set and only then accesses the buckets contents.

# Work-Ahead Set: Post

- Simple implementation of work-ahead set: Circular list or array.
- Main thread posts at next available array slot, in round-robin fashion.
  - If slot already contained a  $(p,s)$  pair, this is returned to the main thread.
  - At the end of the computation, main thread may need to post dummy references to retrieve remaining entries.



# Work-Ahead Set: Read

- Helper thread maintains its own offset  $i$  into the array. Reads entries in circular fashion — *no* coordination with main thread.
  - Helper thread performs a read access to given memory reference. In C:

```
temp += *((int*)p);
```

- If helper thread runs faster than main thread, the same data will be loaded multiple times.
  - If combination  $\langle i, p \rangle$  has been seen before, put thread in spin loop, polling for new entries.

# Work-Ahead Set Entries

- One operation might touch data types larger than a cache line or might need multiple references to execute a single instruction, *e.g.*

```
*a = *b + *c;
```

- Divide execution into three stages.  
But: Too many stages risk intermediate cache eviction.
- Alter structure of work-ahead set entry:  $(p_1, p_2, p_3, s)$ .  
Three times fewer entries, helper thread is supposed to access all three references together.

# Work-Ahead Set Entries

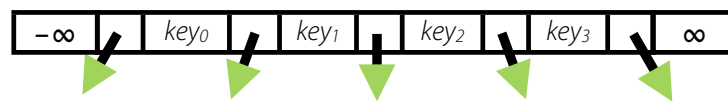
- Use work-ahead set entries with  $M$  memory references each ( $M$ : maximum of references needed per stage in main thread).
  - If an operation needs  $k < M$  entries, patch the remaining  $M - k$  entries with a dummy address  $d$ :  
 $(p_1, p_2, d, d, d, d, s)$
  - No special treatment in helper thread, always access all  $M$  references (reduce branches).
  - Dummy cache line at  $d$  will be accessed often and thus reside in the L1 cache. Negligible overhead.

# Staged Database Operators

- Requires reimplementing of each query operator. Helper thread is general and can be reused.
- Identify stages in operator execution:  
When a memory location  $p_2$  is only known after accessing another memory location  $p_1$  has been read, process  $p_1$  and  $p_2$  in different stages.
- Operators typically perform only simple computations per tuple  
⇒ # of required memory references  $\approx$  operator arity.

# Staged Index Traversal

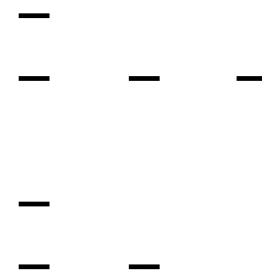
$$key_2 \leq K < key_3$$



- Stage 1:  
On lookup for key  $K$ , traverse B+-tree node and compare with  $key_i$  values. When branch number is known, post  $(child\ node\ address, K)$  to work-ahead set.
  - Stage 2:  
Via post receive  $(child\ node\ address', K')$ . Access that node and resume search for key  $K'$ .
- ⇒ Multiple index probes active “in parallel.”

# Staged Hash Join

- Hash table, chained hash cells  
(= tuple of build table hashed to the associated bucket).
- In hash cell:  $(RID, h(K'))$
- Operator receives  $(RID, K)$  from probe input.
- Perform hash probe in four stages. In each stage:  
(1) compute; (2) issue memory reference for next stage.



# Staged Hash Join

- - -  
- - -  
- -

- Stages:

1. Compute bucket number via hash function  $h(K)$ ,  
post (*hash bucket addr*,  $(h(K),2)$  ). next  
stage
2. Visit hash bucket, post (*first cell addr*,  $(h(K),3)$  ).
3. Visit cell, evaluate  $h(K) = h(K')$ . If equal,  
post (*RID*,  $(K,4)$ ), else post(*next cell addr*,  $(h(K),3)$ ).
4. Visit potentially matching tuple in build relation,  
compare its key with  $K$ . If equal, build result tuple,  
else post(*next cell addr*,  $(h(K),3)$ ).

# Work-Ahead Set Size

- Work-ahead set too small:
  - Insufficient time for helper thread to preload.
  - High probability that main and helper thread access same cache line in work-ahead set. Post of main thread dirties cache line: flush to RAM, reload.
- Work-ahead set too large:
  - Loads performed by helper thread may be evicted from cache before used by main thread.  
Keep size of preloaded data less than (a fraction of) L2 cache size.

# Main / Helper Thread Speed

- Helper thread faster than main thread:
  - Helper thread preloads entire work-ahead set, catches up with main thread. Goes into spin-loop.
  - When main thread advances, helper thread will wake up, also advances one slot, then spin-loops. Same cache line written / read, dirty cache lines, flushing, reloading.
- One solution: Let helper thread process the work-ahead set in a backwards fashion.

# B+–Trees and Cache Performance

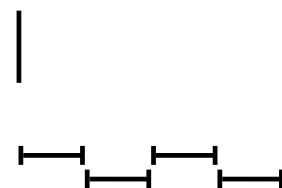
- B+–trees still are the prevailing index structures in database systems.
- The many B+–tree supported operations of a database kernel (selection, access to inner relation of a join) break down into two low-level access patterns:
  1. Descending from root to leaf while searching for a key  $K$ ,
  2. scanning a region of the index leaf level while retrieving the RIDs for a range  $\ell < K \leq r$ .

# B+–Trees and Cache Performance

- Both access patterns effectively follow a linked list of pointers.
  - Each traversal step potentially incurs the cost of page miss (disk-based indexes) or cache miss (memory-based index), since index node size and unit of transfer are aligned.
  - Research has found B+–tree key searches to spend more than 65% of the overall search time waiting for data cache misses.

## B+–Tree Index Node Size

- In-memory, the unit of transfer is two orders of magnitude smaller than for disk-based indexes ( $\approx 64$  byte cache line vs. 4 KB disk page).
  - $\Rightarrow$  In-memory indexes are considerably deeper.
  - As tree height determines number of cache misses during key search, try to increase fan-out (branching factor):
  - One approach: (Almost) No pointers in inner index nodes, instead point to node group. Double fan-out.

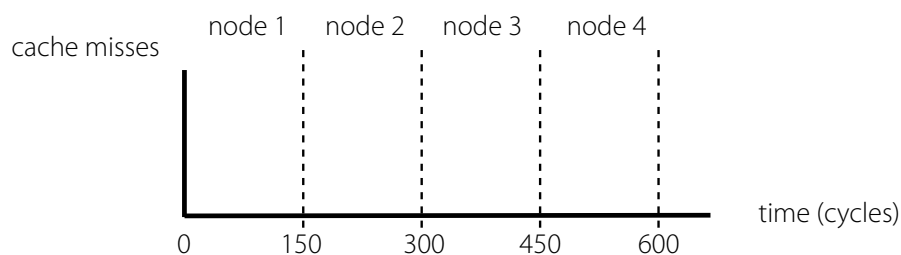


# Prefetching B+-Trees

- Remember that for a modern CPU and memory subsystem, it is *not* the number of cache misses but the amount of exposed miss latency that counts.
- Use prefetching in B+-trees to limit exposed miss latency.
  - Searching and scanning in B+-trees pose a challenging problem since both constitute *pointer-chasing problems*.
  - Adapt the original B+-tree concept to enable prefetching that reaches sufficiently far ahead.

# Wider Index Nodes

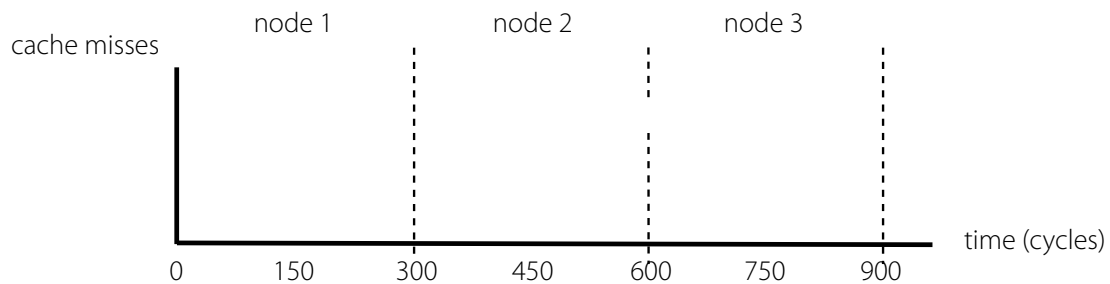
- Expect one cache miss while moving down a level of the tree. How to reduce tree depth?
  - B+-tree of 1000 keys, node size 64 bytes (cache line), key size = pointer size = RID size = 4 bytes, B+-tree will contain  $\geq 4$  levels. Miss latency 150 cy:



- But: Simply making nodes wider than transfer unit can hurt performance.

# Wider Index Nodes

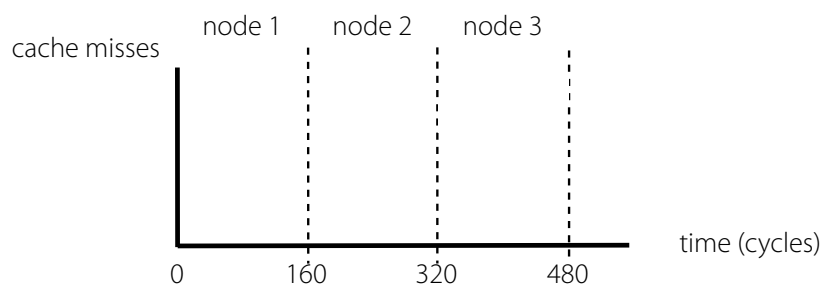
- Double the node width. One node occupies two cache lines (128 bytes). New tree depth: 3 levels.
- Cache misses during B+-tree key search:



- Search now suffers six cache misses (instead of four), decreasing search performance.

# Wide Index Nodes Plus Prefetching

- Prefetch second half of each two-cache-line-wide node already upon node entry.
  - Second half of node at known (contiguous) address.
- Assume: Memory can serve two back-to-back cache misses with a 10 cycle delay ( $10 \text{ cy} \ll 150 \text{ cy}$ ).



# Minor Modifications to B+–Tree Algorithms

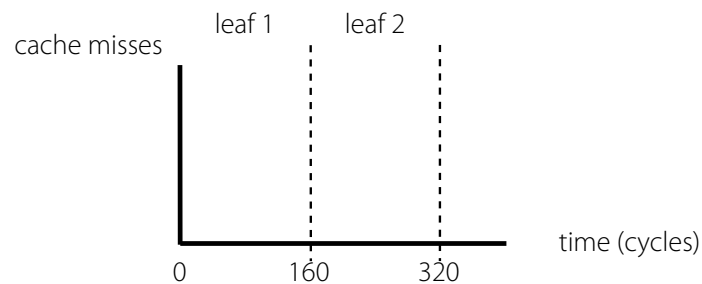
- Search:  
Upon node entry, before starting binary search, prefetch all cache lines that comprise a node.
- Insert:  
Perform index search, brings root-to-leaf path into cache. Cache misses only for newly allocated nodes.
- Deletion:  
Perform index search, root-to-leaf path in cache. Redistribute only on completely empty nodes. In that case, prefetch sibling node (not on prefetched path).

# Prefetching Index Scans

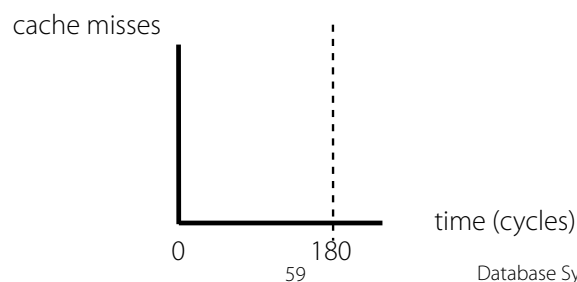
- Given start and end key, perform index search for start key to obtain start leaf node.
  - Then let the scan follow *next leaf* pointers, visiting leaf nodes in order.
  - Copy scanned RIDs (or tuples) into result buffer.
  - Stop scan if end key reached or result buffer full (may resume scan later on).
- Studies found 84% data cache stall time during index scans.

# Wide Leaf Nodes

- Apply prefetching idea to yield wider leaf nodes:



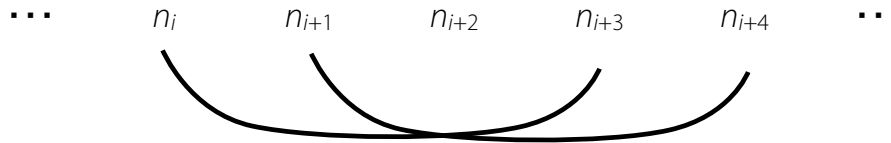
- But index scans lead to a longer sequence of node accesses. Goal: Try to prefetch farther ahead:



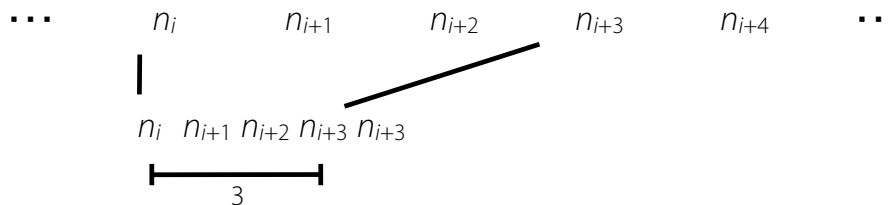
# Pointer-Chasing Problem

- Assume that three leaf nodes worth of computation are needed to hide miss latency.
  - When node  $n_i$  is visited, a prefetch of node  $n_{i+3}$  should be initiated.  
Following the pointer chain through nodes  $n_{i+1}$  and  $n_{i+2}$  is no option.
- Possible solutions:
  1. Data linearization — only for read-only workloads.
  2. Jump pointers (embed prefetch address in node).

# Jump Pointers

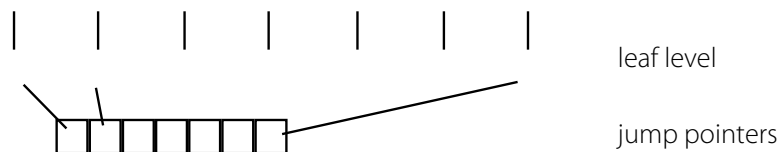


- Pull jump pointers out of nodes into separate jump pointer array. Added flexibility:
  - Adjust prefetching distance easily (adapt to changing performance conditions, model different latencies, e.g., database host or memory vs. disk):



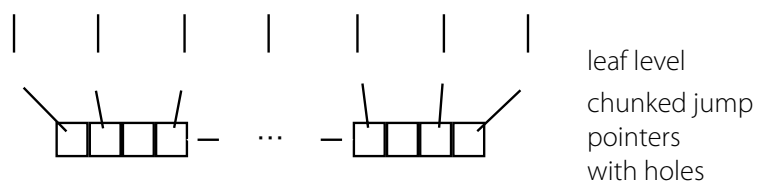
# Updating Jump Pointers

B+-tree



- B+-tree insertion may lead to node split at leaf level: Shift jump pointer array, update back pointers.

B+-tree



# Internal Jump Pointers

- Avoid B+-tree external jump pointer array. Instead reuse pointers from leaf parent level.
- Chain leaf parents in order to support rapid prefetching (overhead: one pointer per node).
- No back pointers needed: remember branching number in leaf parent during the search for start key.

