



# CS 764: Topics in Database Management Systems

## Lecture 3: Radix Join

Xiangyao Yu

9/14/2021

# Today's Paper: Radix Join

## Database Architecture Optimized for the new Bottleneck: Memory Access

Peter Boncz\*  
Data Distilleries B.V.  
Amsterdam · The Netherlands  
P.Boncz@ddi.nl

Stefan Manegold · Martin Kersten  
CWI  
Amsterdam · The Netherlands  
{S.Manegold,M.Kersten}@cwi.nl

### Abstract

In the past decade, advances in speed of commodity CPUs have far out-paced advances in memory latency. Main-memory access is therefore increasingly a performance bottleneck for many computer applications, including database systems. In this article, we use a simple scan test to show the severe impact of this bottleneck. The insights gained are translated into guidelines for database architecture; in terms of both data structures and algorithms. We discuss how vertically fragmented data structures optimize cache performance on sequential data access. We then focus on equi-join, typically a random-access operation, and introduce radix algorithms for partitioned hash-join. The performance of these algorithms is quantified using a detailed analytical model that incorporates memory access cost. Experiments that validate this model were performed on the Monet database system. We obtained exact statistics on events like TLB misses, L1 and L2 cache misses, by using hardware performance counters found in modern CPUs. Using our cost model, we show how the carefully tuned memory access pattern of our radix algorithms make them perform well, which is confirmed by experimental results.

\*This work was carried out when the author was at the University of Amsterdam, supported by SION grant 612-23-431  
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

### 1 Introduction

Custom hardware – from workstations to PCs – has been experiencing tremendous improvements in the past decades. Unfortunately, this growth has not been equally distributed over all aspects of hardware performance and capacity. Figure 1 shows that the speed of commercial microprocessors has been increasing roughly 70% every year, while the speed of commodity DRAM has improved by little more than 50% over the past decade [Mow94]. Part of the reason for this is that there is a direct tradeoff between capacity and speed in DRAM chips, and the highest priority has been for increasing capacity. The result is that from the perspective of the processor, memory has been getting slower at a dramatic rate. This affects all computer systems, making it increasingly difficult to achieve high processor efficiencies.

Three aspects of memory performance are of interest: bandwidth, latency, and address translation. The only way to reduce effective memory latency for appli-

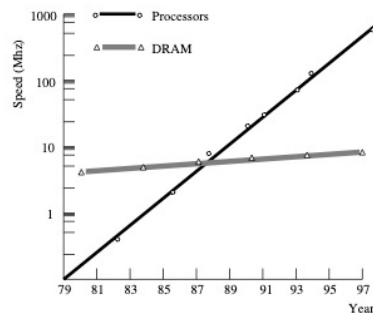


Figure 1: Hardware trends in DRAM and CPU speed

# Agenda

---

Hardware background

In-memory partitioned hash join

Radix join

Experimental results

Radix join vs. non-partition hash join

Column-store and encoding

# Agenda

---

## **Hardware background**

In-memory partitioned hash join

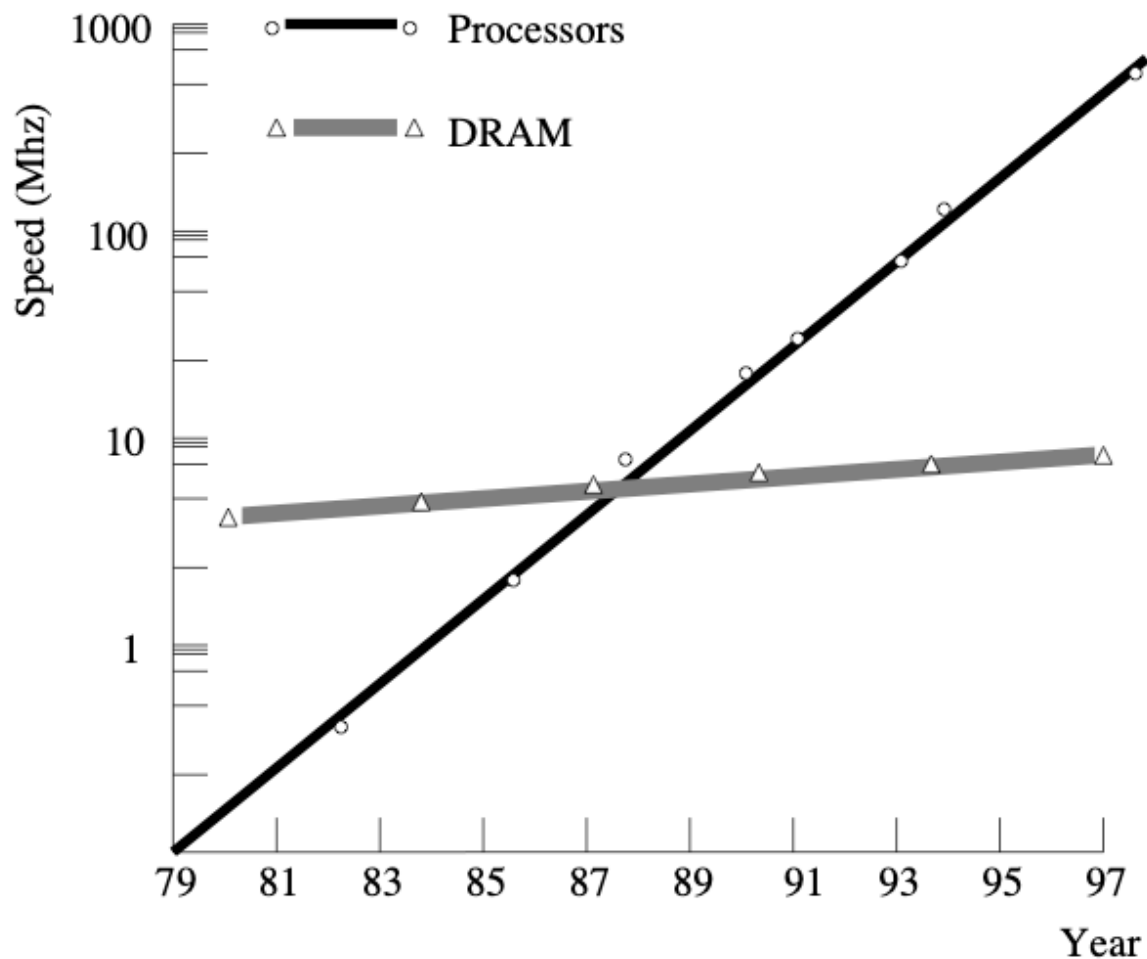
Radix join

Experimental results

Radix join vs. non-partition hash join

Column-store and encoding

# Memory Wall



The growth of memory speed is slower than the growth of CPU speed

- Latency
- Bandwidth

Figure 1: Hardware trends in DRAM and CPU speed

# Memory/Cache Hierarchy

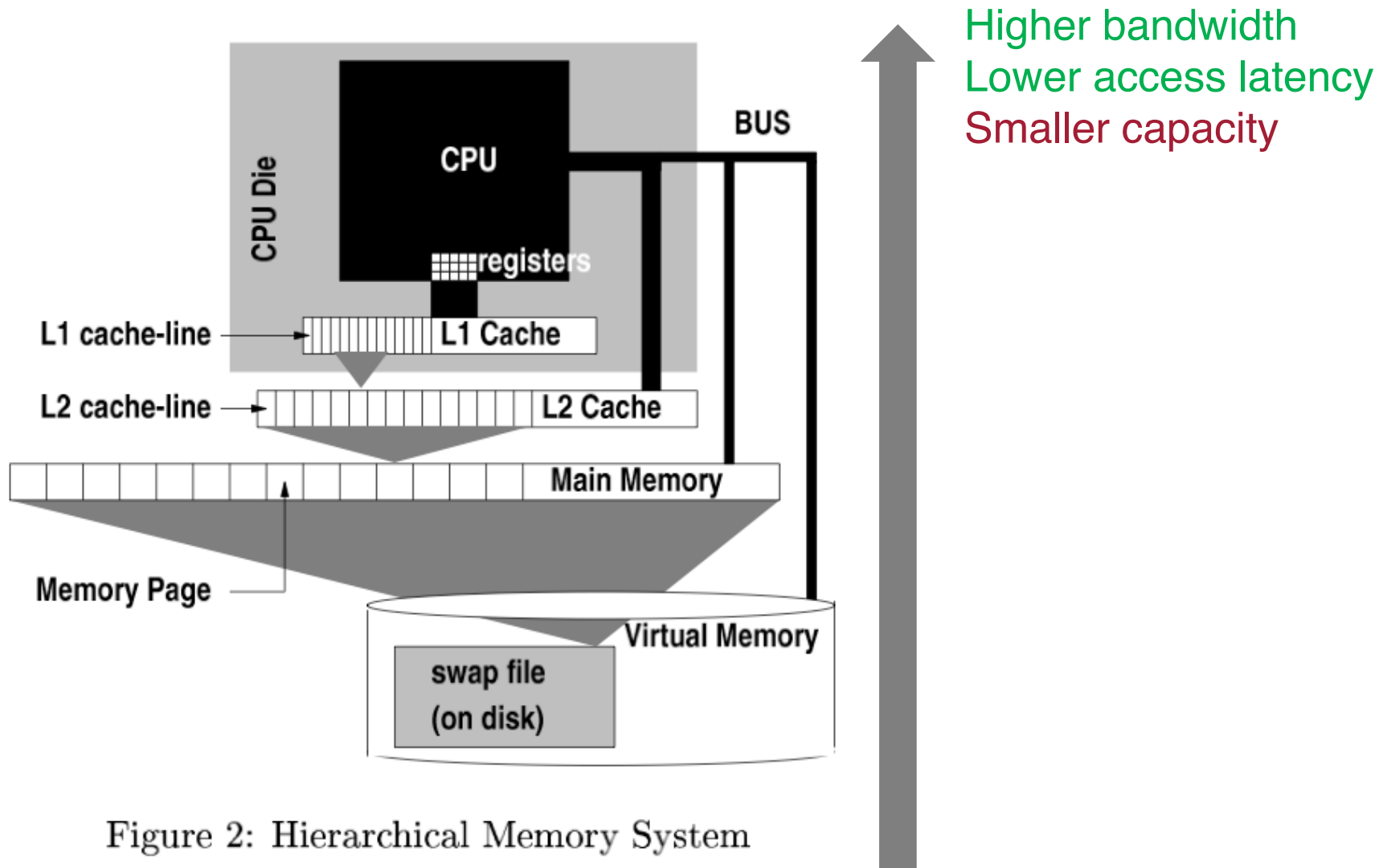
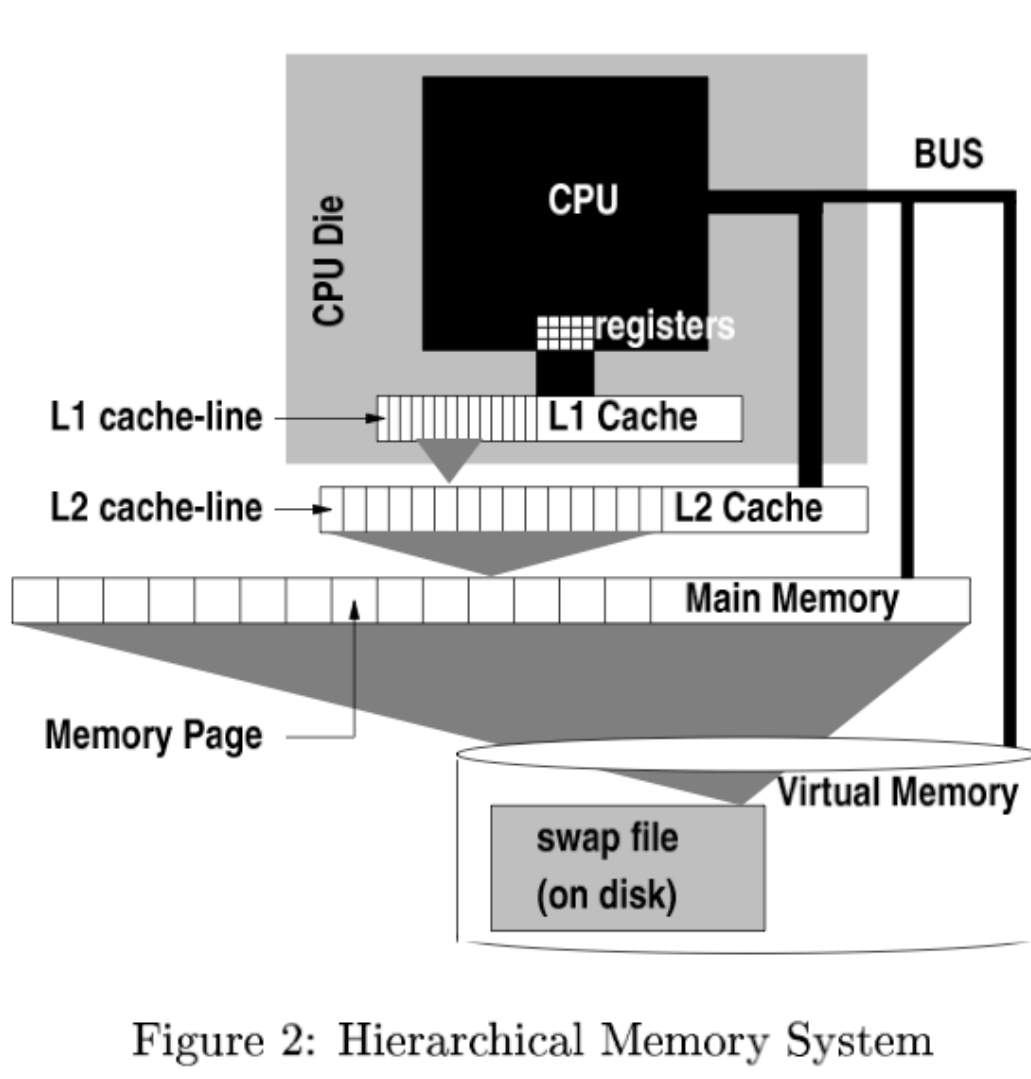


Figure 2: Hierarchical Memory System

# Memory/Cache Hierarchy



Higher bandwidth  
Lower access latency  
Smaller capacity

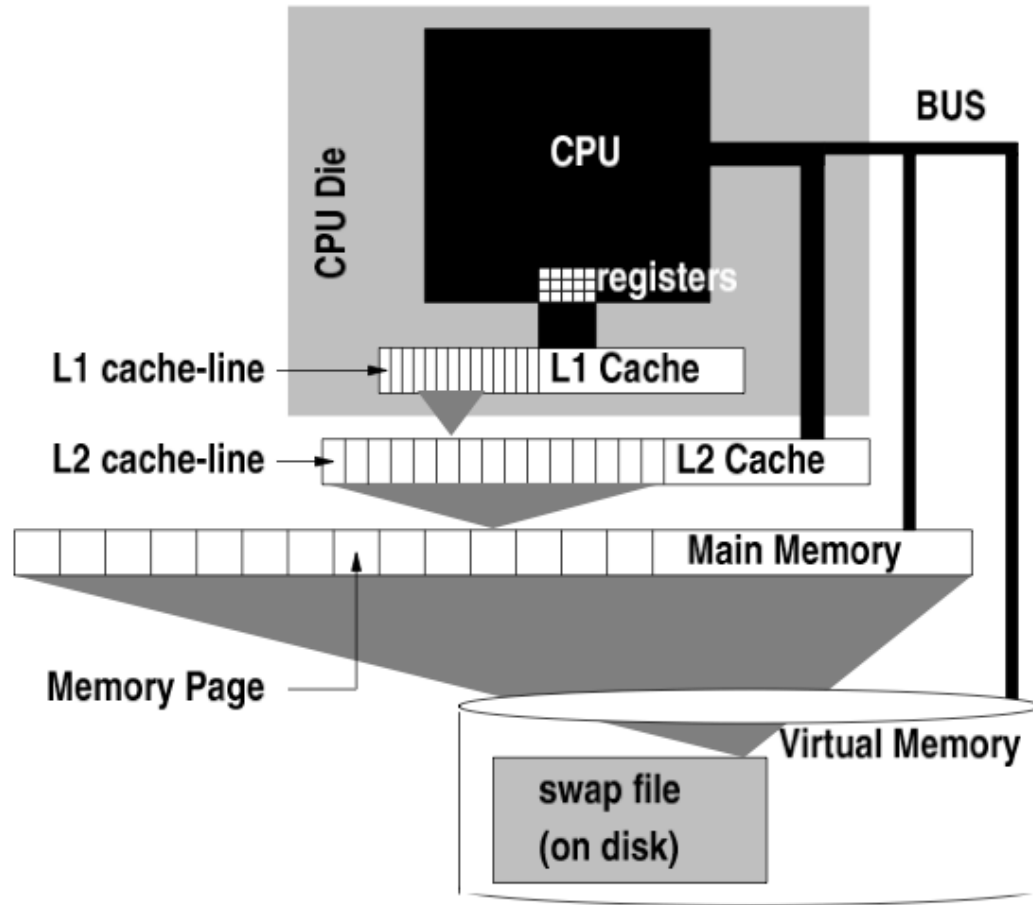
Optimizing join in DRAM/Disk system

– GRACE hash join

Optimizing join in SRAM/DRAM system?

Figure 2: Hierarchical Memory System

# Optimizing Join in Main-Memory DBMS



**Intuitive solution:** Partition tables into shards that fit in SRAM cache  
– Like GRACE hash join

Figure 2: Hierarchical Memory System



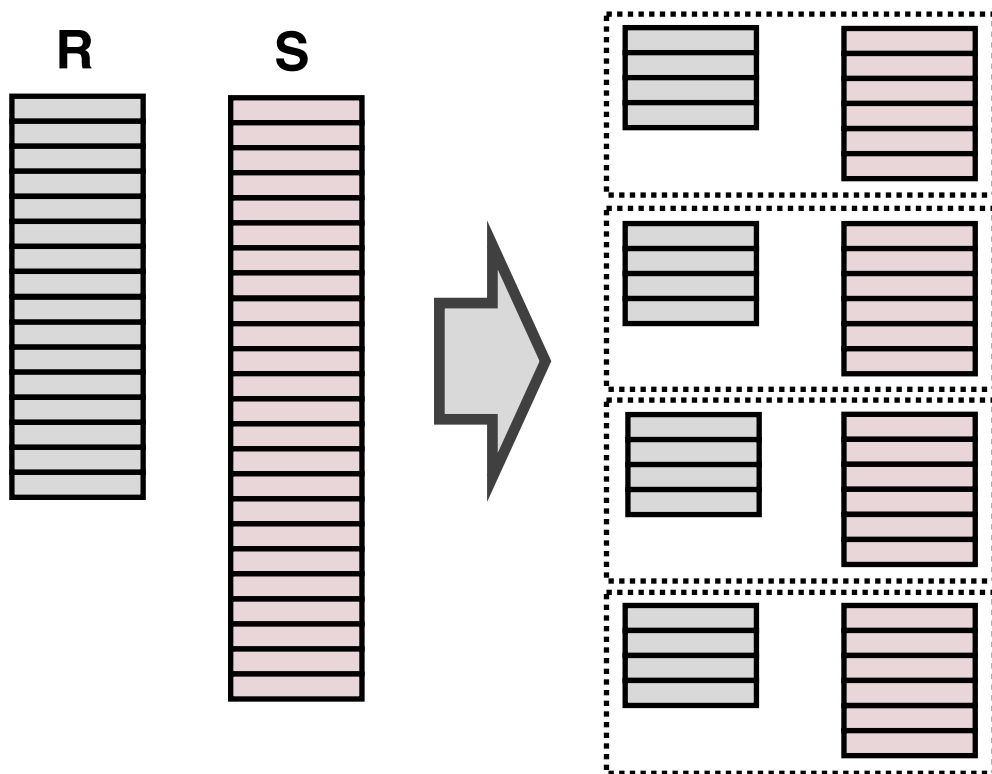
# Recap: GRACE Hash Join

---

Phase 1: Partition both R and S into pairs of k shards

– Each shard of R fits in CPU cache

Phase 2: Separately join each pairs of partitions



# Group Discussion

---

In some modern in-memory DBMSs, the entire database can fit in memory. In such a system, can similar optimizations be applied to on-chip SRAM caches vs. DRAM? What are the key challenges compared to a DRAM vs. Disk setting?

# Group Discussion

---

In some modern in-memory DBMSs, the entire database can fit in memory. In such a system, can similar optimizations be applied to on-chip SRAM caches vs. DRAM? What are the key challenges compared to a DRAM vs. Disk setting?

- Software does not have full control of CPU cache contents
- Disk access granularity is a block; DRAM access granularity is a cacheline
- CPU cache has very limited capacity

# Optimizing Join in Main-Memory DBMS

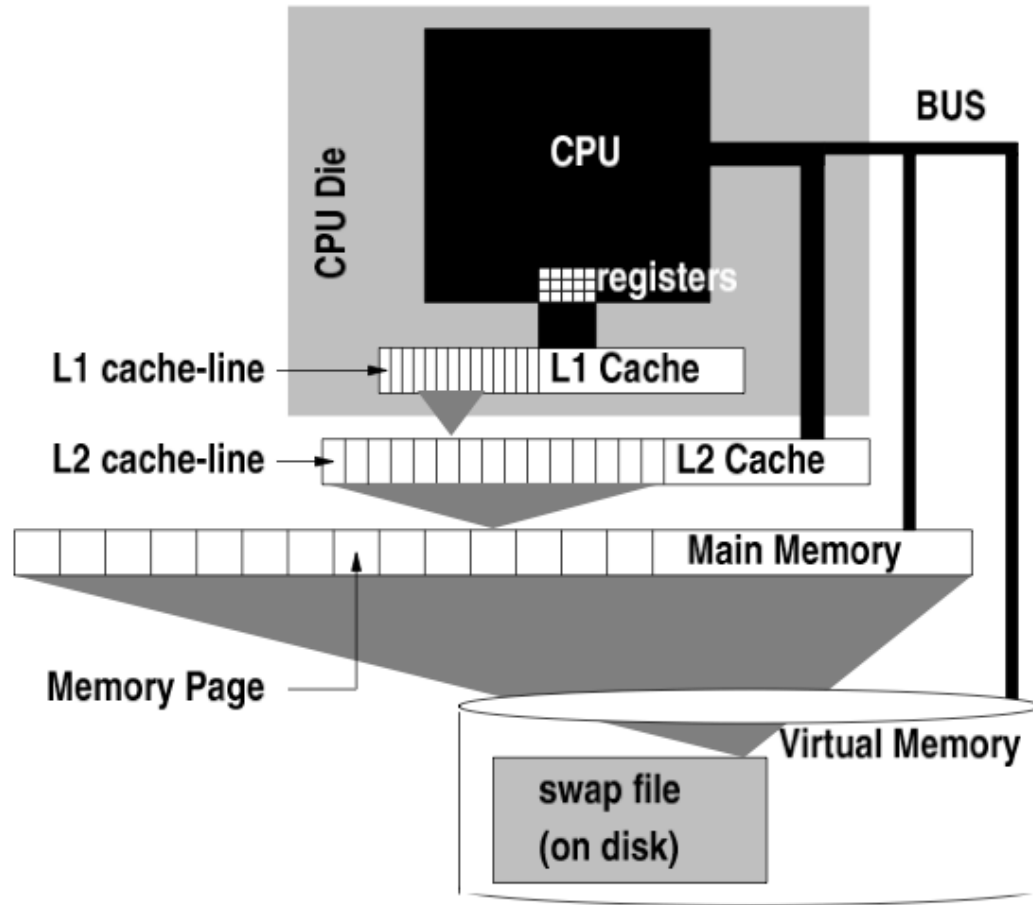


Figure 2: Hierarchical Memory System

**Intuitive solution:** Partition tables into shards that fit in SRAM cache

- Like GRACE hash join

**Challenges:**

- TLB becomes a performance bottleneck if too many partitions exist
- Determine the memory layout of data partitions (e.g., fragmentation)

# Agenda

---

Hardware background

**In-memory partitioned hash join**

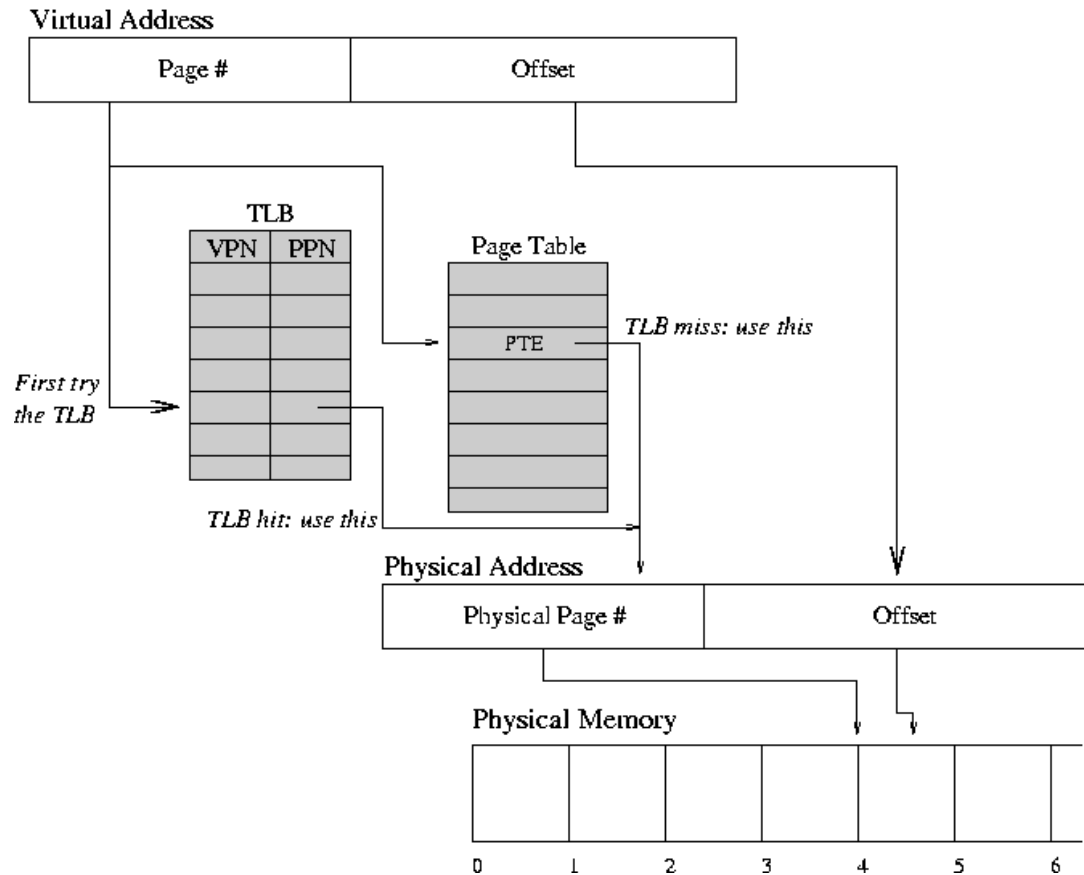
Radix join

Experimental results

Radix join vs. non-partition hash join

Column-store and encoding

# Translation Lookaside Buffer (TLB)

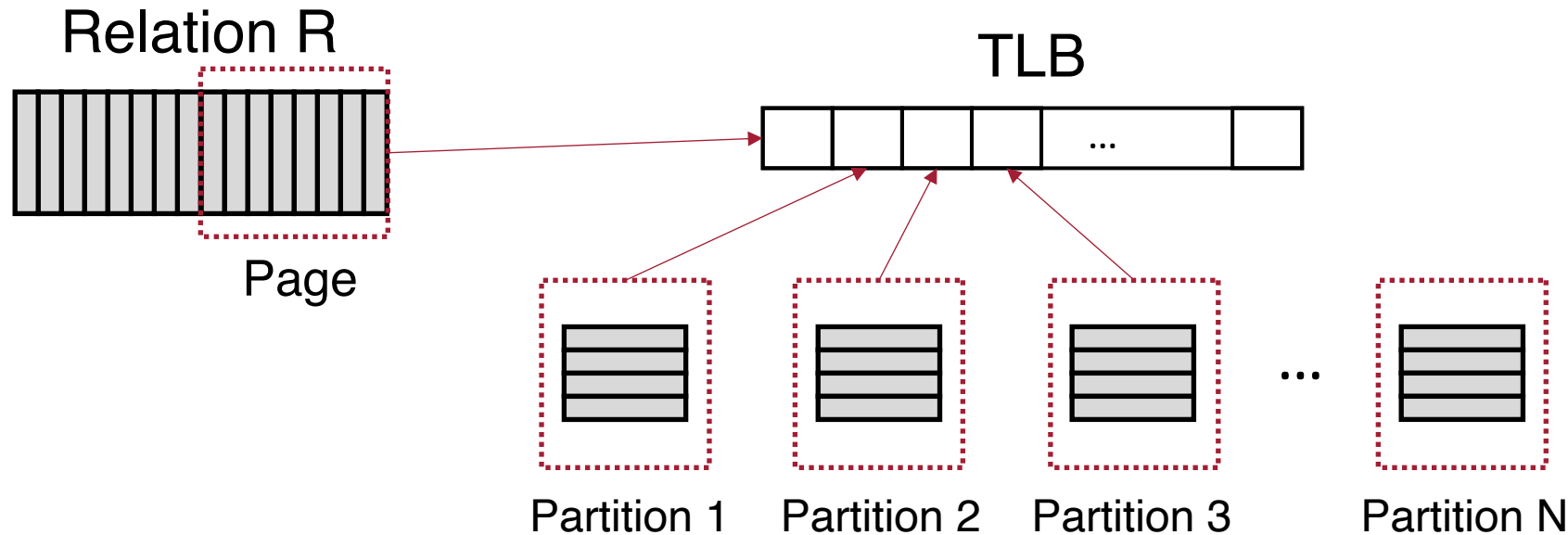


- A cache of OS page table to accelerate virtual address to physical address translation
- TLB hit has no cost
  - TLB miss requires an expensive page table walk

TLB has a small number of entries

source: <http://pages.cs.wisc.edu/~bart/537/lecturenotes/s17.html>

# # of Partitions vs. TLB size



If the number of partitions is greater than the number of TLB entries, the system experiences **TLB thrashing**, i.e., many accesses lead to TLB misses

# Thrashing

---

**TLB thrashing:** Number of accessed pages (i.e., number of partitions) is greater than the number of TLB entries in hardware

**Cache thrashing:** Number of accessed cachelines (i.e., number of partitions) is greater than the cache capacity

**Page thrashing** (in last lecture): Number of accessed pages (i.e., number of partitions) is greater than the memory capacity



# Optimizing Join in Main-Memory DBMS

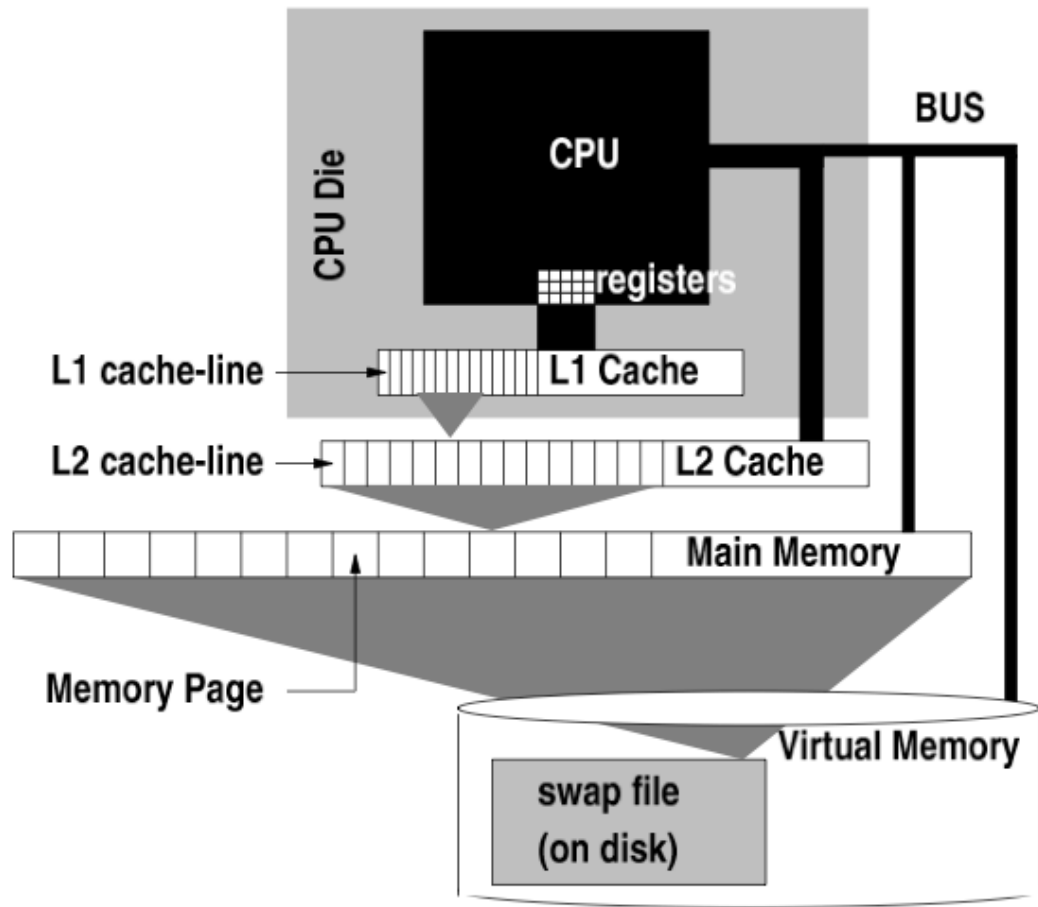


Figure 2: Hierarchical Memory System

Intuitive solution: Partition tables into shards that fit in SRAM cache

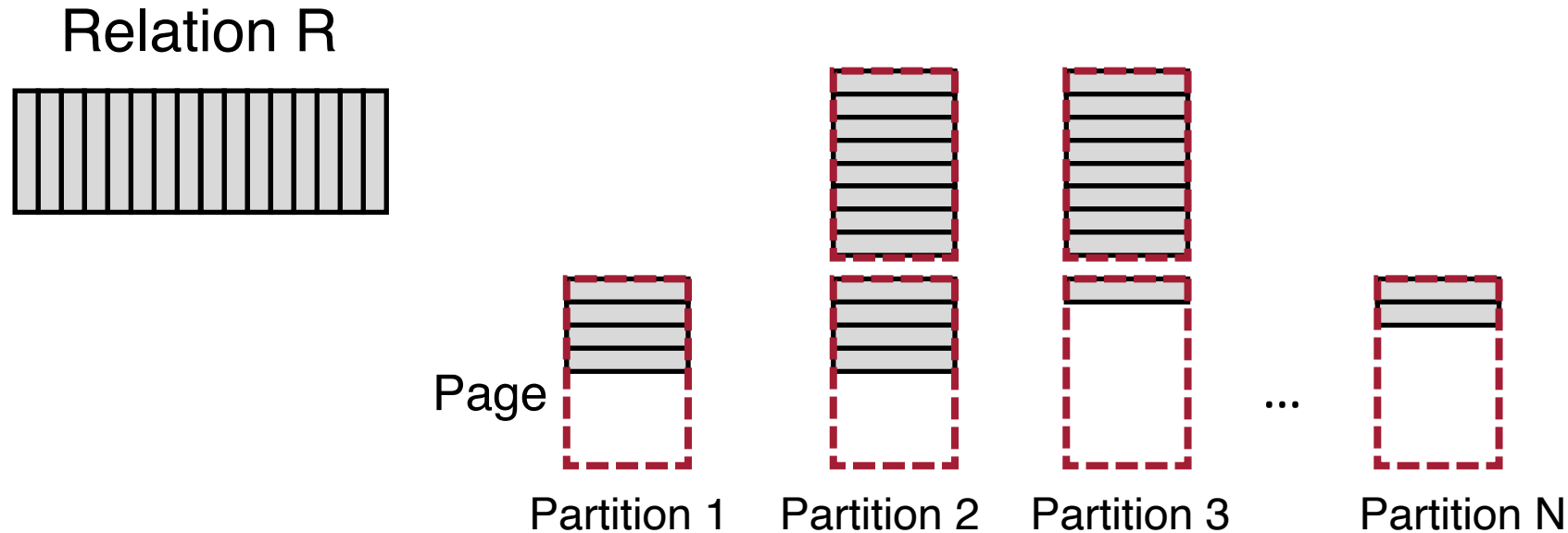
Challenges:

- **TLB becomes a performance bottleneck if too many partitions exist**
- Determine the memory layout of data partitions (e.g., fragmentation)

Do not have too many partitions per round of partitioning. Limiting factor includes cache size and TLB size.

# Fragmentation

---



How to track **location and size** for different partitions?

- Frequent memory allocation (e.g., malloc) is expensive
- Loss of memory capacity due to fragmentation
- Problem becomes worse if multiple passes of partitioning is needed

# Agenda

---

Hardware background

In-memory partitioned hash join

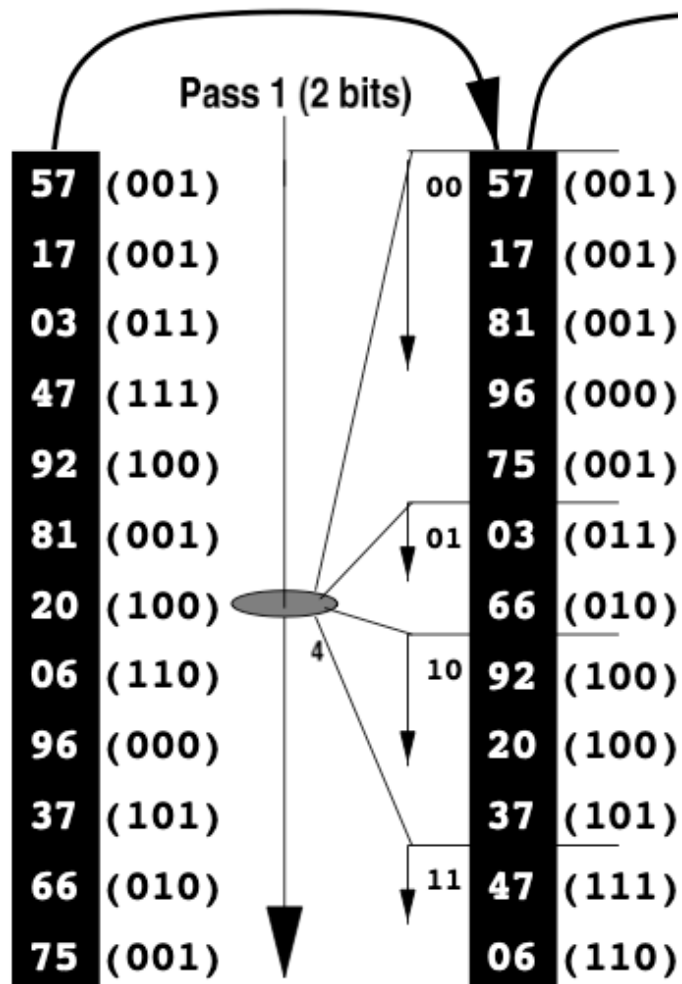
**Radix join**

Experimental results

Radix join vs. non-partition hash join

Column-store and encoding

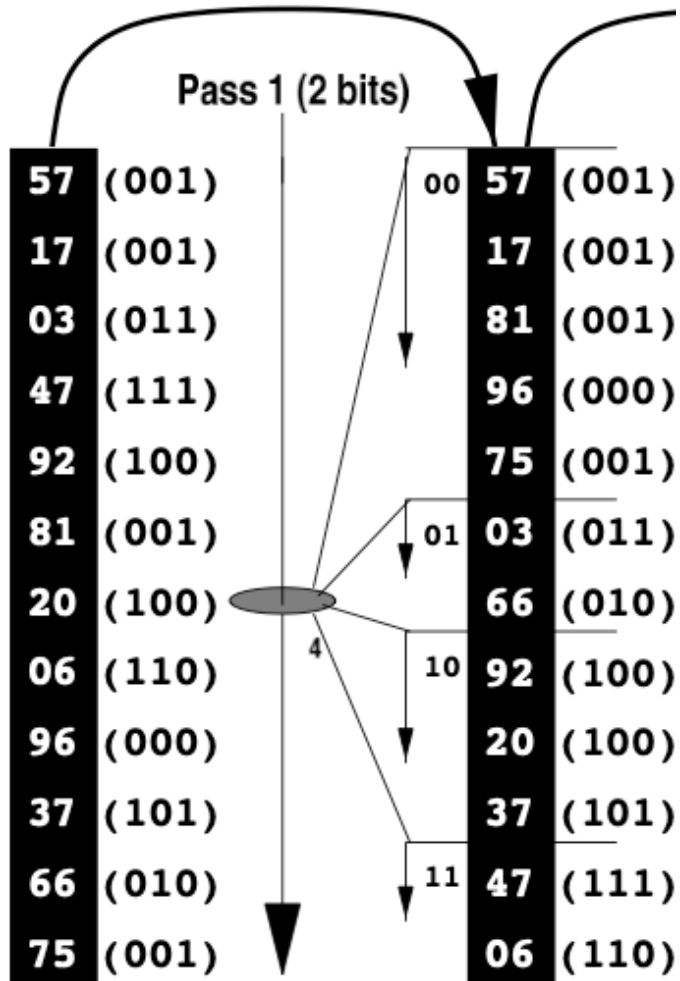
# Radix Partitioning



Cluster on the **lower B bits** of the **integer hash-value** of the partition key

- For pass  $p$ , use  $B_p$  bits for partitioning
- Start with left most bits

# Radix Partitioning



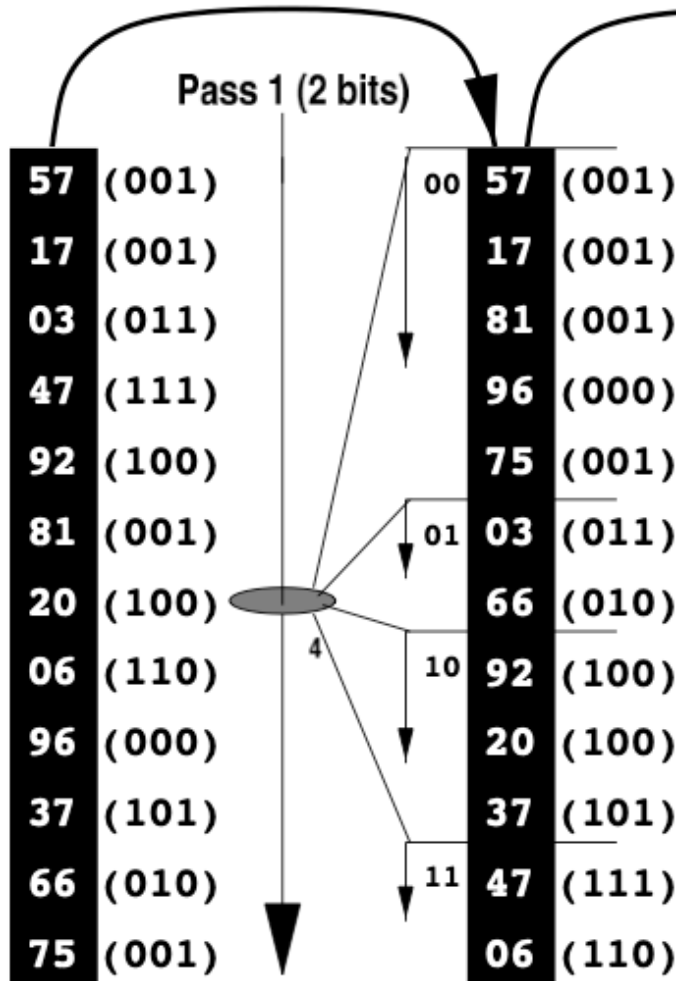
Cluster on the **lower B bits** of the **integer hash-value** of the partition key

- For pass  $p$ , use  $B_p$  bits for partitioning
- Start with left most bits

The output array of Radix partitioning has identical structure as the input array

- No complex memory allocation
- No fragmentation

# Radix Partitioning



Cluster on the **lower B bits** of the **integer hash-value** of the partition key

- For pass  $p$ , use  $B_p$  bits for partitioning
- Start with left most bits

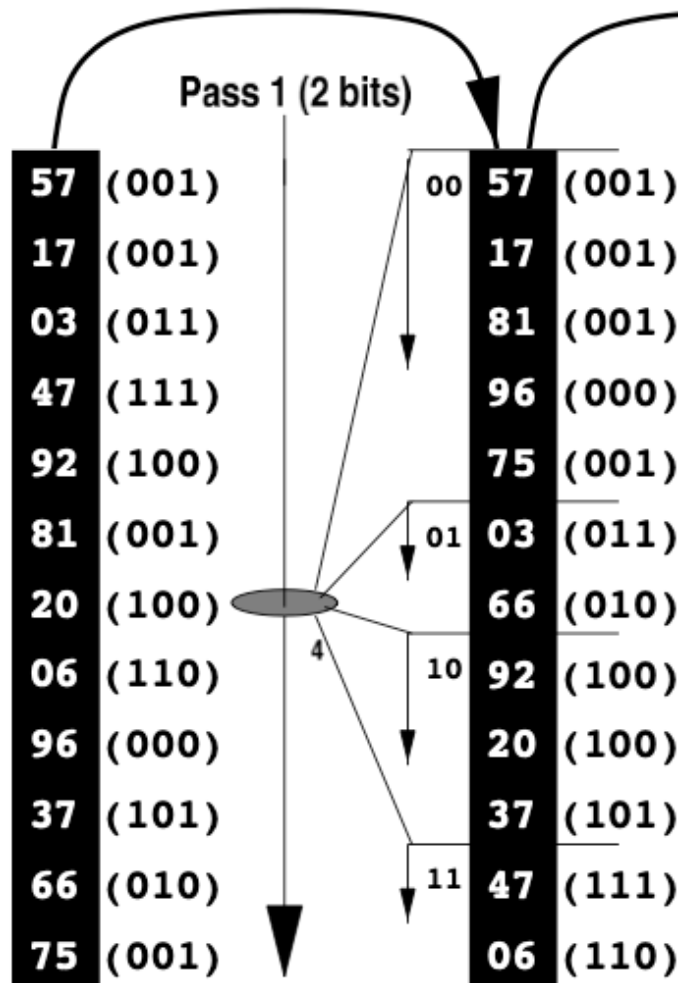
The output array of Radix partitioning has identical structure as the input array

- No complex memory allocation
- No fragmentation

Q: How to know where to write in the output array? (e.g., 47 in the example)

- Need to scan the array twice; first time collect size per partition

# Radix Partitioning



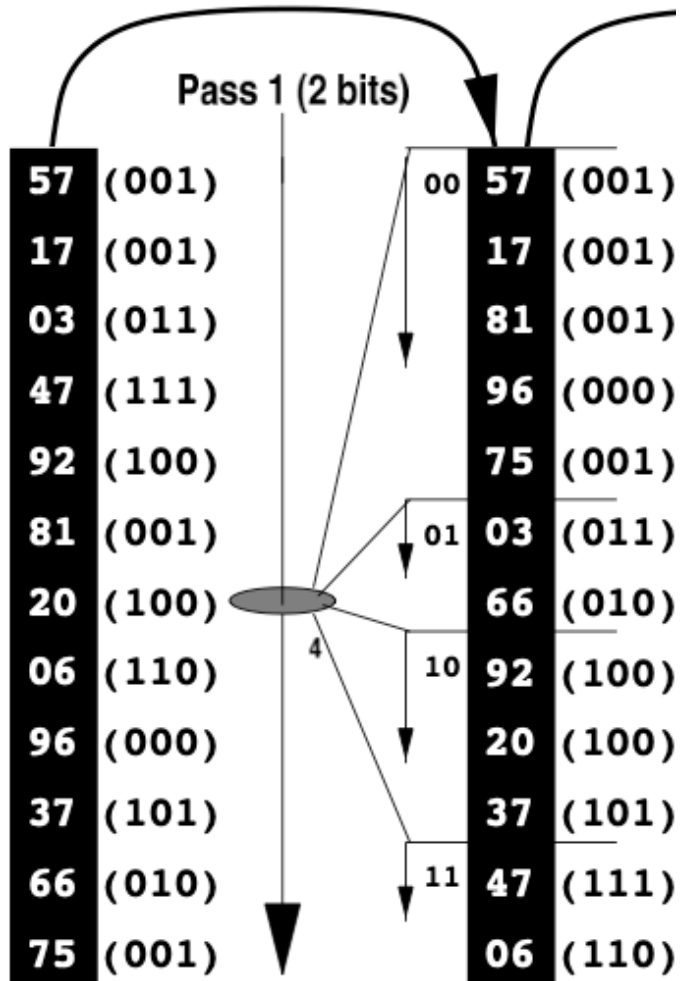
## First scan

- "00": 5 records
- "01": 2 records
- "10": 3 records
- "11": 2 records

Q: How to know where to write in the output array? (e.g., 47 in the example)

- Need to scan the array twice; first time collect size per partition

# Radix Partitioning



## First scan

- "00": 5 records
- "01": 2 records
- "10": 3 records
- "11": 2 records

Write location in output buffer

- "00": entry 0
- "01": entry 5
- "10": entry 5 + 2
- "11": entry 5 + 2 + 3

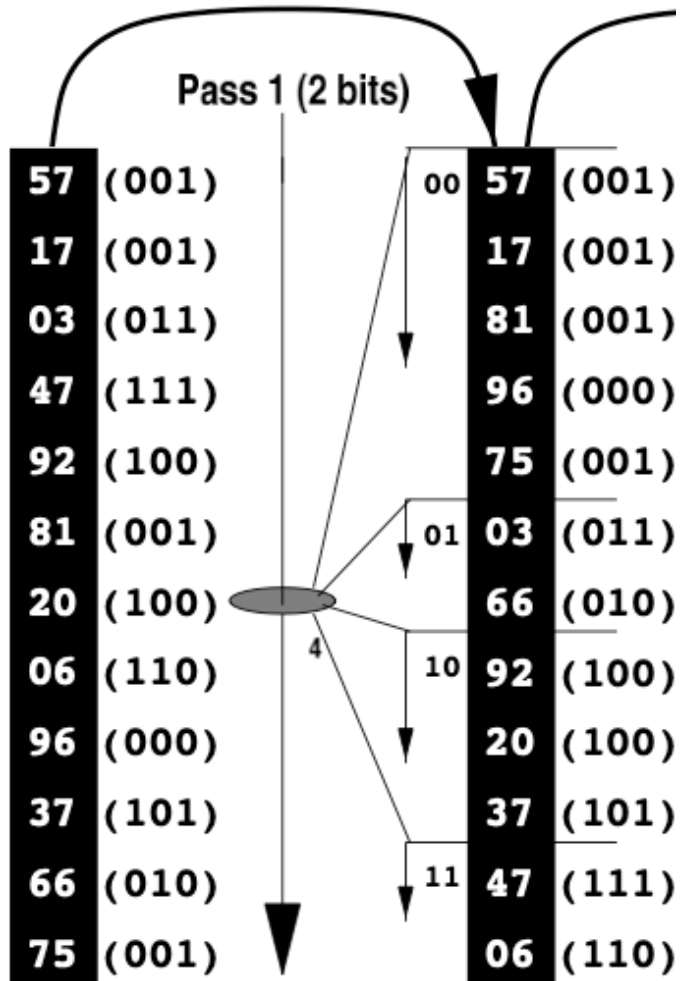
Prefix Sum

Q: How to know where to write in the output array? (e.g., 47 in the example)

- Need to scan the array twice; first time collect size per partition



# Radix Partitioning



## First scan

- "00": 5 records
- "01": 2 records
- "10": 3 records
- "11": 2 records

Write location in output buffer

- "00": entry 0
- "01": entry 5
- "10": entry 5 + 2
- "11": entry 5 + 2 + 3

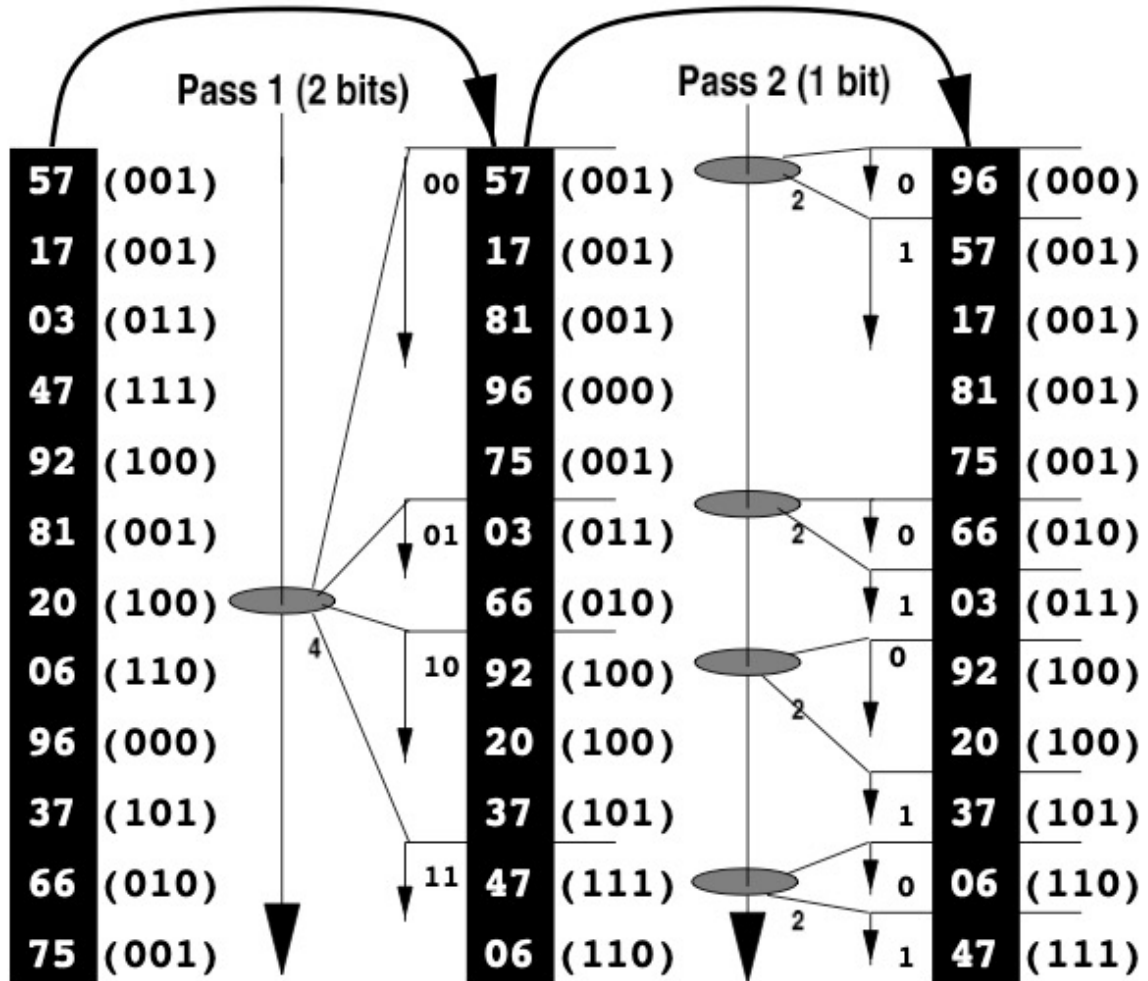
Prefix Sum

**Second scan:** write to corresponding location in the output buffer

Q: How to know where to write in the output array? (e.g., 47 in the example)

- Need to scan the array twice; first time collect size per partition

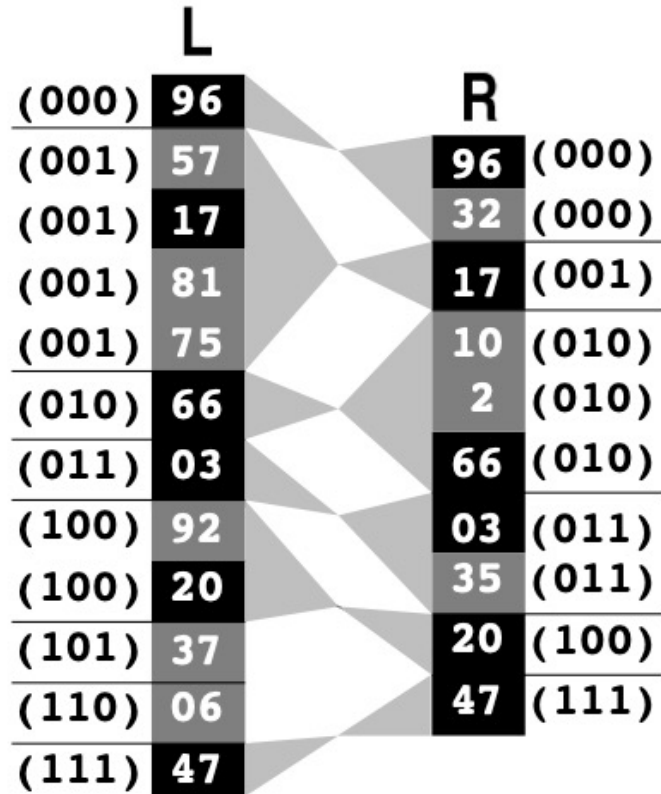
# Radix Partitioning



Fully clustering  $B$  bits may require multiple passes

Number of partitions per pass is bounded by TLB and cache size

# Join



Similar to GRACE hash join, join the corresponding partitions from the two relations

Can use either **hash join** or **nested-loop join**

## Discussion Question:

Can we use sort merge join for the two relations?

# Agenda

---

Hardware background

In-memory partitioned hash join

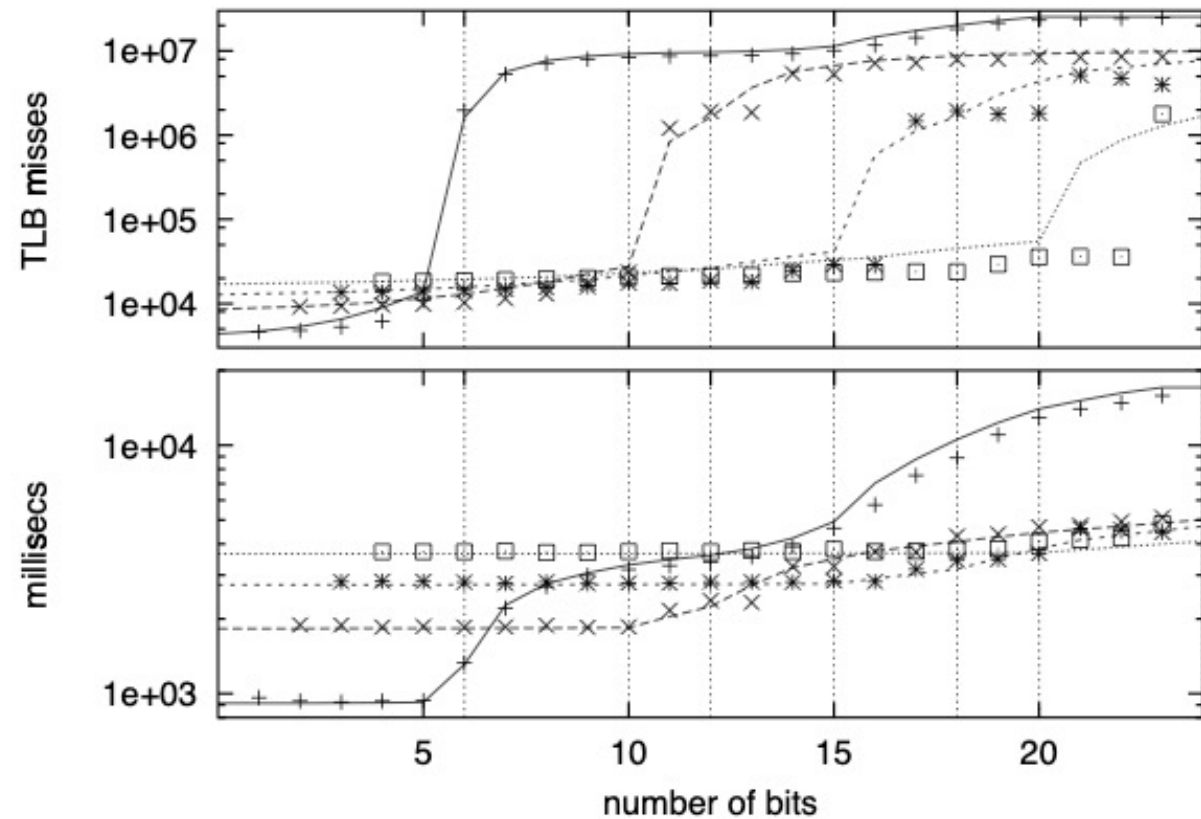
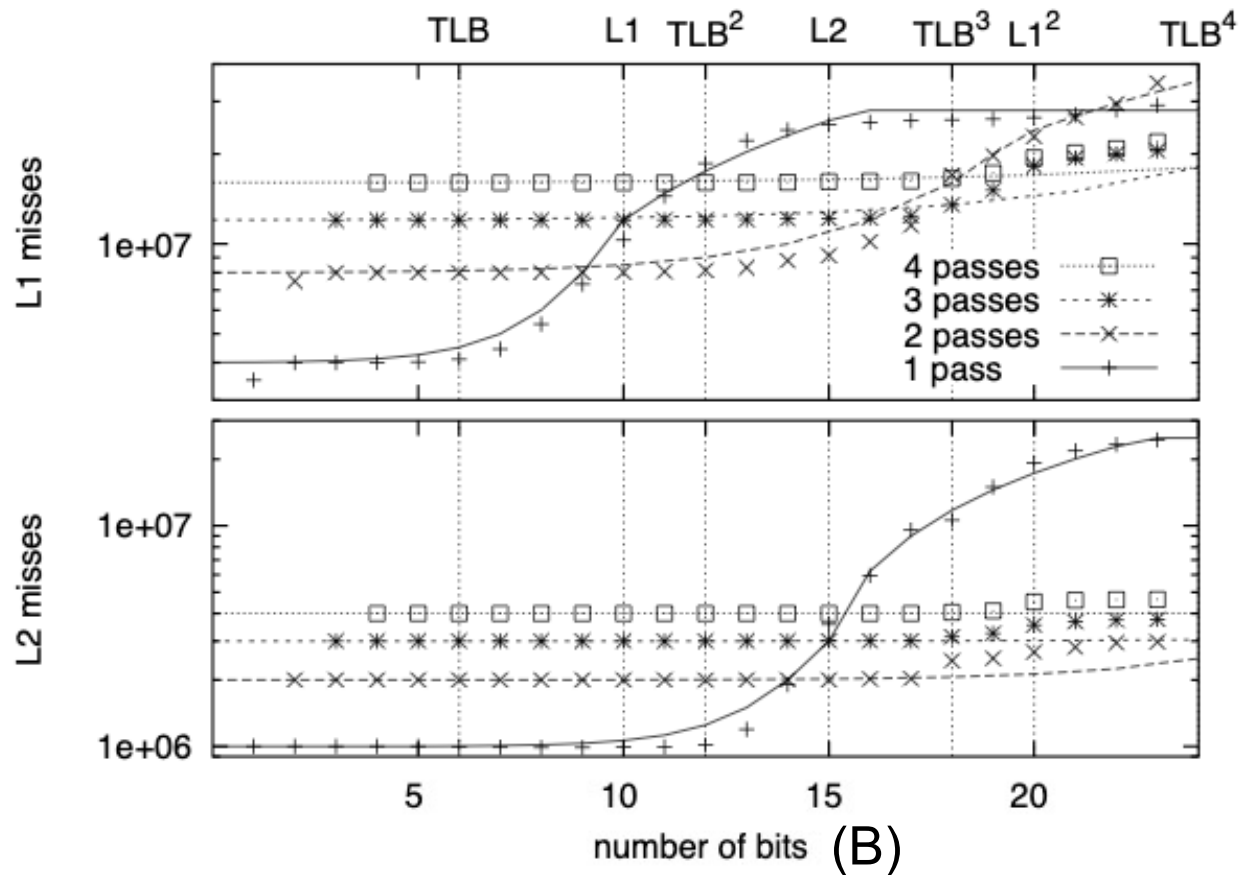
Radix join

**Experimental results**

Radix join vs. non-partition hash join

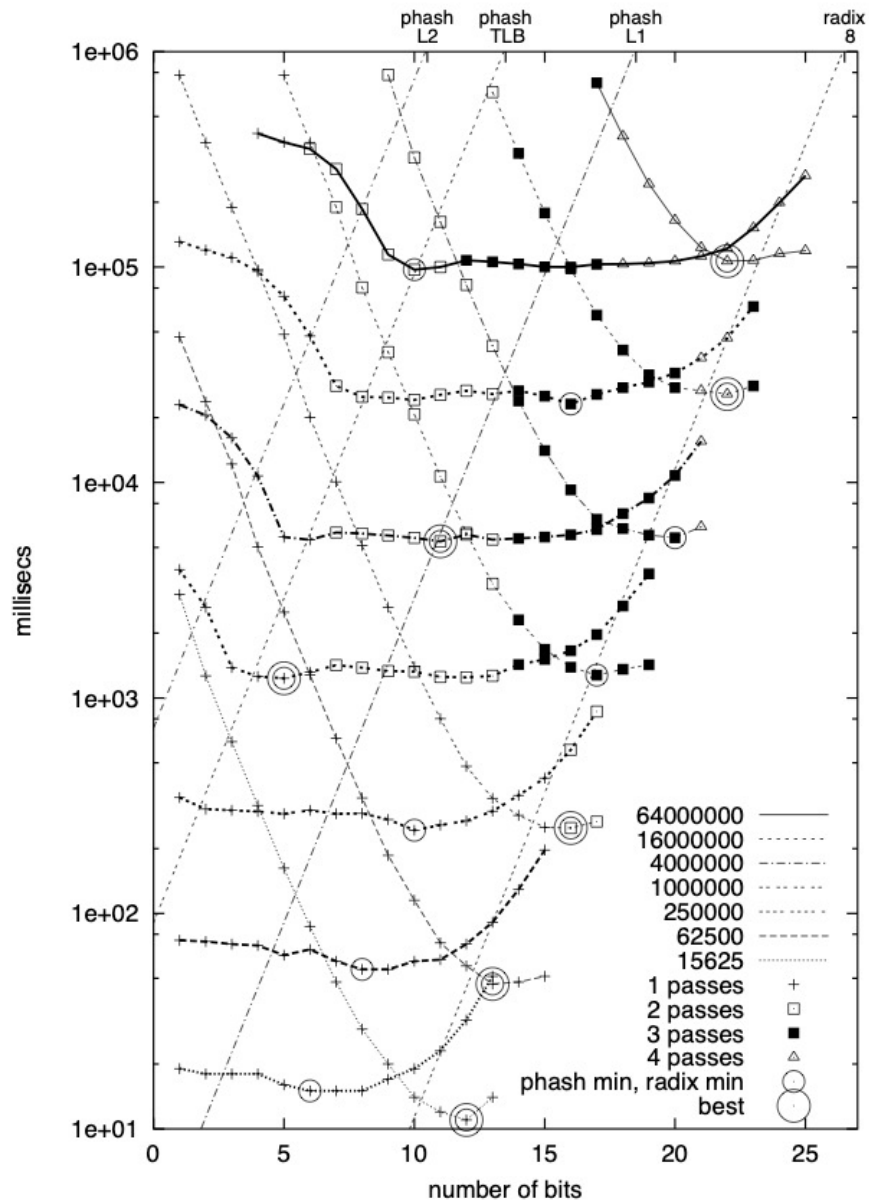
Column-store and encoding

# Evaluation: Radix Clustering



The machine's TLB has 64 entries

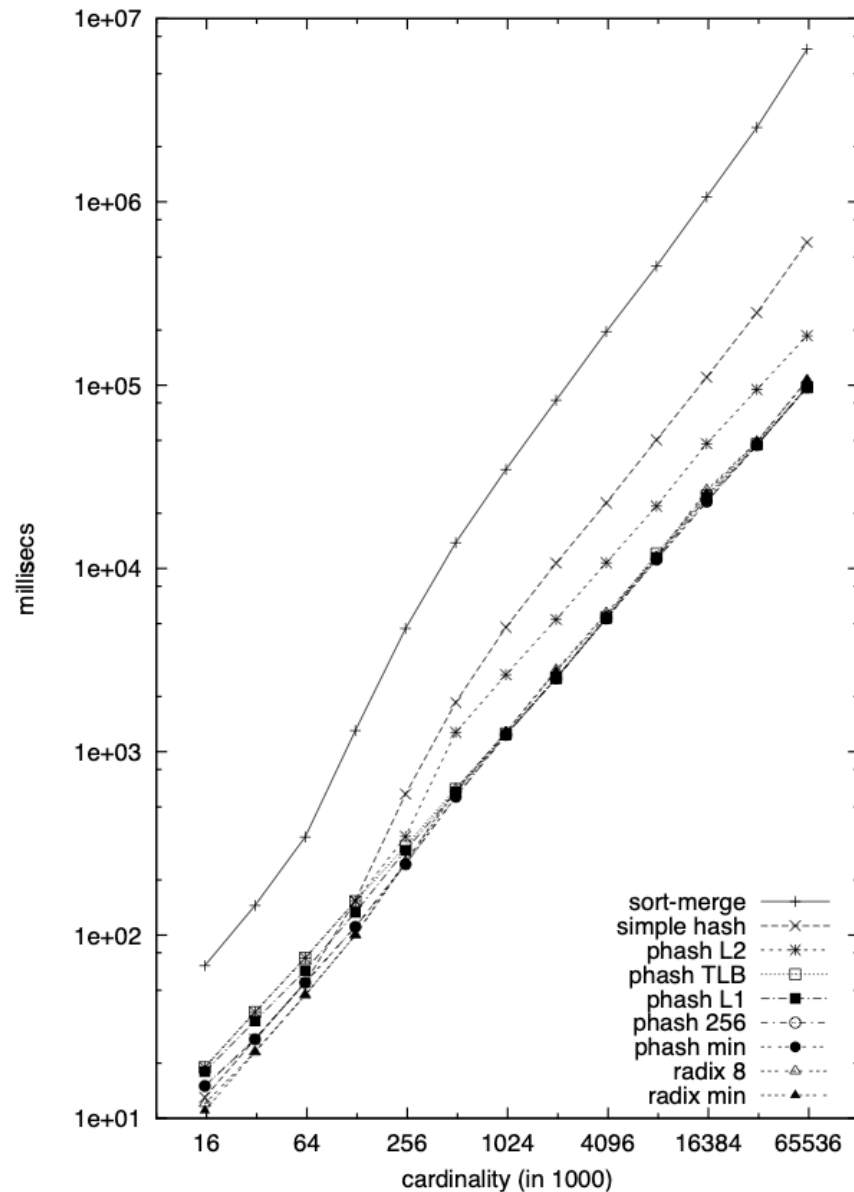
# Evaluation: Join Performance



Nested-loop join prefers small partitions

Hash-join achieves similar performance for a range of partition sizes

# Evaluation: Overall



Sort-merge < Simple hash < phash L2 < phash TLB and the rest

# Agenda

---

Hardware background

In-memory partitioned hash join

Radix join

Experimental results

**Radix join vs. non-partition hash join**

Column-store and encoding



# A Different View Point

## Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs

Spyros Blanas Yinan Li Jignesh M. Patel  
University of Wisconsin–Madison  
{sblanas, yinan, jignesh}@cs.wisc.edu

### ABSTRACT

The focus of this paper is on investigating efficient hash join algorithms for modern multi-core processors in main memory environments. This paper dissects each internal phase of a typical hash join algorithm and considers different alternatives for implementing each phase, producing a family of hash join algorithms. Then, we implement these main memory algorithms on two radically different modern multi-processor systems, and carefully examine the factors that impact the performance of each method.

Our analysis reveals some interesting results – a very simple hash join algorithm is very competitive to the other more complex methods. This simple join algorithm builds a shared hash table and does not partition the input relations. Its simplicity implies that it requires fewer parameter settings, thereby making it far easier for query optimizers and execution engines to use it in practice. Furthermore, the performance of this simple algorithm improves dramatically as the skew in the input data increases, and it quickly starts to outperform all other algorithms. Based on our results, we propose that database implementers consider adding this simple join algorithm to their repertoire of main memory join algorithms, or adapt their methods to mimic the strategy employed by this algorithm, especially when joining inputs with skewed data distributions.

### Categories and Subject Descriptors

H.2.4. [Database Management]: Systems—Query processing, Relational databases

### General Terms

Algorithms, Design, Performance

### Keywords

hash join, multi-core, main memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD '11, June 12–16, 2011, Athens, Greece.  
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...\$10.00.

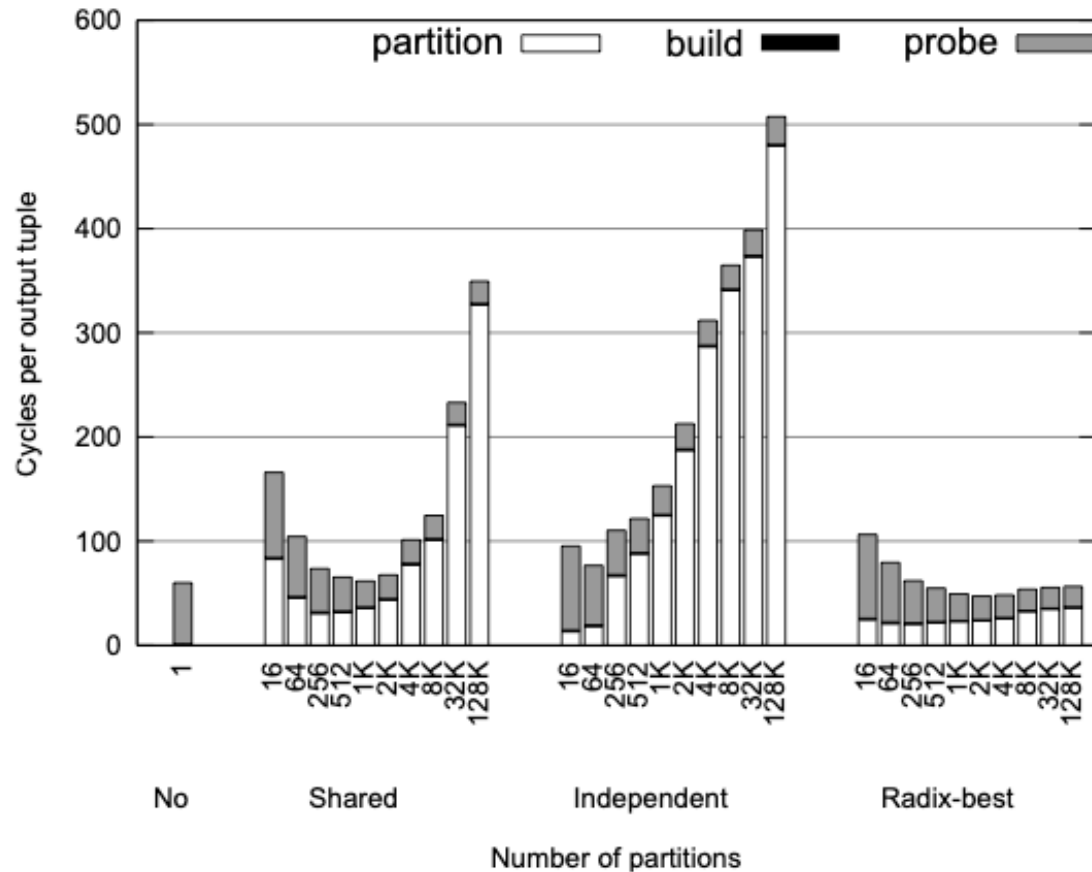
12 years later

Multicore processors

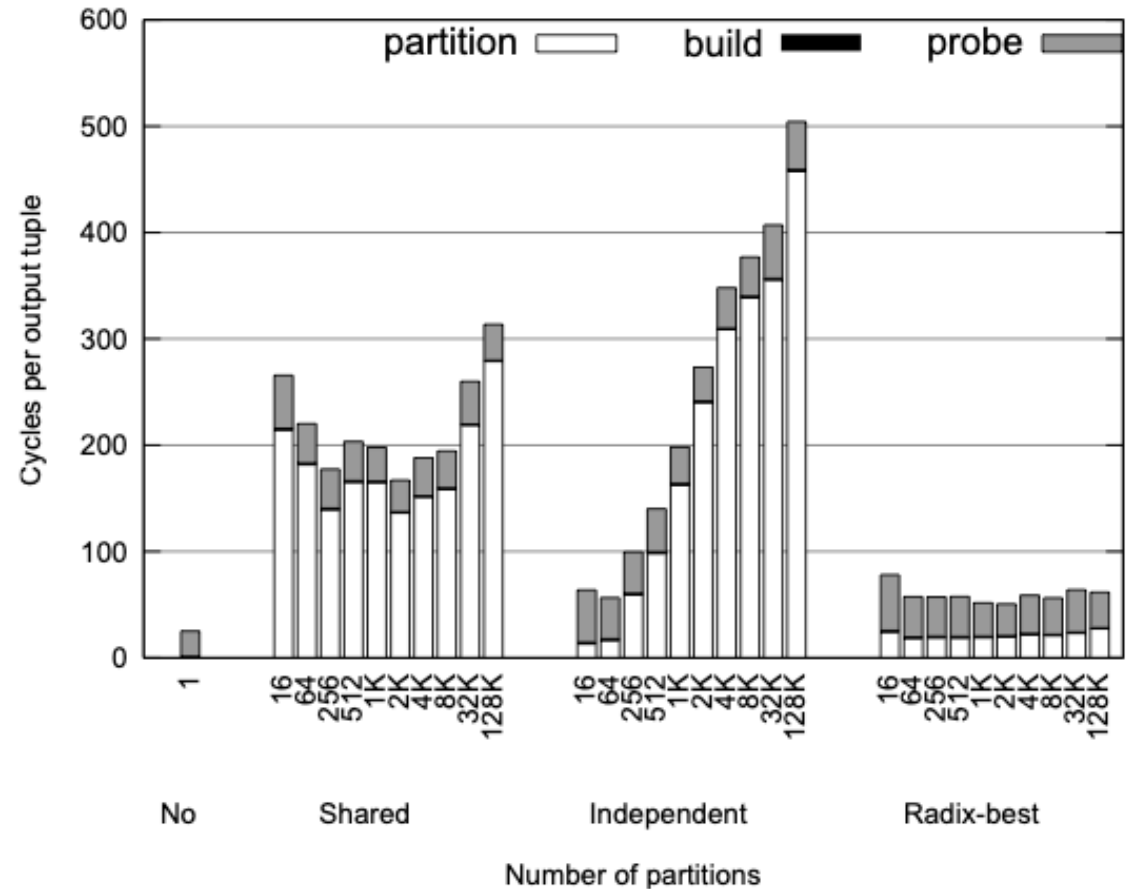
Two design considerations

- minimizing the number of processor cache misses => **Radix Join**
- minimizing processor synchronization costs => **No partition hash join**

# Evaluation on Multicore



**Uniform dataset**



**Highly skewed dataset**

Important to minimize synchronization overhead in multicore processors

# Agenda

---

Hardware background

In-memory partitioned hash join

Radix join

Experimental results

Radix join vs. non-partition hash join

**Column-store and encoding**

# Other Topics

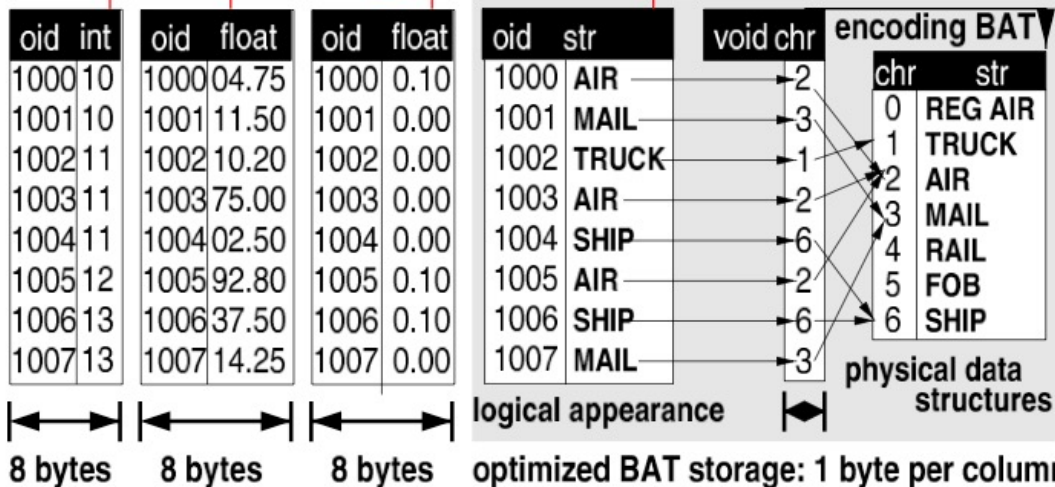
"Item" Table

| supporder | part | price | discnt | qty | tax  | flag | status | shipmode | date1 | date2 | comment |
|-----------|------|-------|--------|-----|------|------|--------|----------|-------|-------|---------|
| • 10      | •    | 04.75 | •      | •   | 0.10 | •    | •      | AIR      | •     | •     | •       |
| • 10      | •    | 11.50 | •      | •   | 0.00 | •    | •      | MAIL     | •     | •     | •       |
| • 11      | •    | 10.20 | •      | •   | 0.00 | •    | •      | TRUCK    | •     | •     | •       |
| • 11      | •    | 75.00 | •      | •   | 0.00 | •    | •      | AIR      | •     | •     | •       |
| • 11      | •    | 02.50 | •      | •   | 0.00 | •    | •      | SHIP     | •     | •     | •       |
| • 12      | •    | 92.80 | •      | •   | 0.10 | •    | •      | AIR      | •     | •     | •       |
| • 13      | •    | 37.50 | •      | •   | 0.10 | •    | •      | SHIP     | •     | •     | •       |
| • 13      | •    | 14.25 | •      | •   | 0.00 | •    | •      | MAIL     | •     | •     | •       |

int int int float float int float char(1) int varchar date date date char(27)

width of relational tuple ≈ 80 bytes

vertical fragmentation in Monet



## Column-store for analytical databases

- Stonebraker, Mike, et al. *C-store: a column-oriented DBMS*. VLDB 2005

# Other Topics

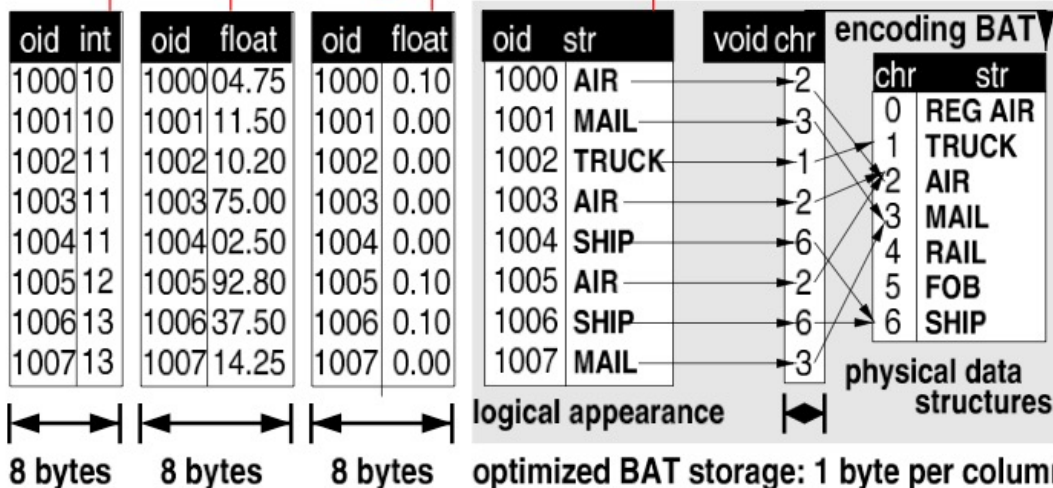
"Item" Table

| supporder | part | price   | discnt | qty | tax    | flag | status | shipmode | date1 | date2 | comment |
|-----------|------|---------|--------|-----|--------|------|--------|----------|-------|-------|---------|
| • 10      | • 10 | • 04.75 | •      | •   | • 0.10 | •    | •      | AIR      | •     | •     | •       |
| • 10      | • 10 | • 11.50 | •      | •   | • 0.00 | •    | •      | MAIL     | •     | •     | •       |
| • 11      | • 11 | • 10.20 | •      | •   | • 0.00 | •    | •      | TRUCK    | •     | •     | •       |
| • 11      | • 11 | • 75.00 | •      | •   | • 0.00 | •    | •      | AIR      | •     | •     | •       |
| • 11      | • 11 | • 02.50 | •      | •   | • 0.00 | •    | •      | SHIP     | •     | •     | •       |
| • 12      | • 12 | • 92.80 | •      | •   | • 0.10 | •    | •      | AIR      | •     | •     | •       |
| • 13      | • 13 | • 37.50 | •      | •   | • 0.10 | •    | •      | SHIP     | •     | •     | •       |
| • 13      | • 13 | • 14.25 | •      | •   | • 0.00 | •    | •      | MAIL     | •     | •     | •       |

int int int float float int float char(1) int varchar date date date char(27)

width of relational tuple ≈ 80 bytes

vertical fragmentation in Monet



## Column-store for analytical databases

- Stonebraker, Mike, et al. *C-store: a column-oriented DBMS*. VLDB 2005

## Dictionary encoding

- Many other encoding/compression schemes exist. E.g., bit-packing, delta encoding, RLE, etc.

# Radix Join – Comments and Q/A

---

Radix join ensures tuples with same join key belong to same cluster?

Radix join assumes attributes stored as compact integer array?

Disadvantage of radix join?

Why having a shared hash table efficient for skewed data?

Common approach to use analytical model?

How to pick best parameters? (configurations vary across machines)

CPU speed improvement is also slowing down now.

Can radix join make use of modern hierarchical memory systems?

Core idea of radix-join portable to other operators?

# Group Discussion

---

We want to join three tables,  $S \bowtie R \bowtie T$ . Assume  $S$  is large but  $R$  and  $T$  are relatively small (but larger than CPU cache). Assume the two joins are on different join keys. Would you use **non-partitioned hash join** or **radix join** for this query? Please justify your choice.

# Before Next Lecture

---

Submit review for

Hong-Tai Chou, David DeWitt, [An Evaluation of Buffer Management Strategies for Relational Database Systems.](#)  
Algorithmica, 1986