Exercise Session 4

Data Processing on Modern Hardware
263-3502-00L — Fall Semester 2012

Cagri Balkesen
cagri.balkesen@inf.ethz.ch

Department of Computer Science ETH Zurich, Switzerland

11 October 2012

Assigment 3
→ Partitioned Hash Join

The Classic Hash Join

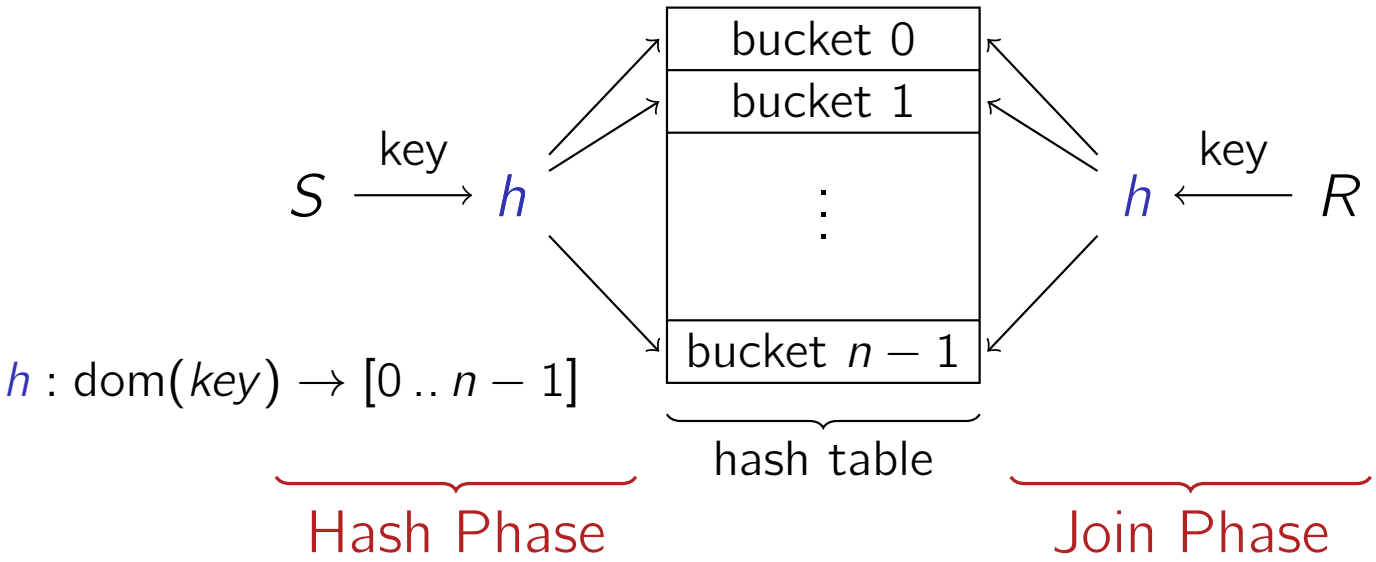
To compute $R \bowtie S$,

1. **Build a hash table** on the "inner" join relation S. } Hash Phase
2. **Scan** the "outer" relation R and **probe** into the hash table for each tuple $r \in R$. } Join Phase

```
1 Function: hash_join(R, S)
  // Hash Phase
2 foreach tuple s ∈ S do
3     insert s into hash table H ;
  // Join Phase
4 foreach tuple r ∈ R do
5     probe H and append matching tuples to result ;
```

Hash Join and Caches

A virtue of hashing is the $\mathcal{O}(1)$ access for every tuple.



But that's also its Achilles' heel:

- Bucket access is inherently "random."
- Hash table access is going to become **expensive** when the size of the hash table exceeds the size of the cache.

Assignment 3 : Overview

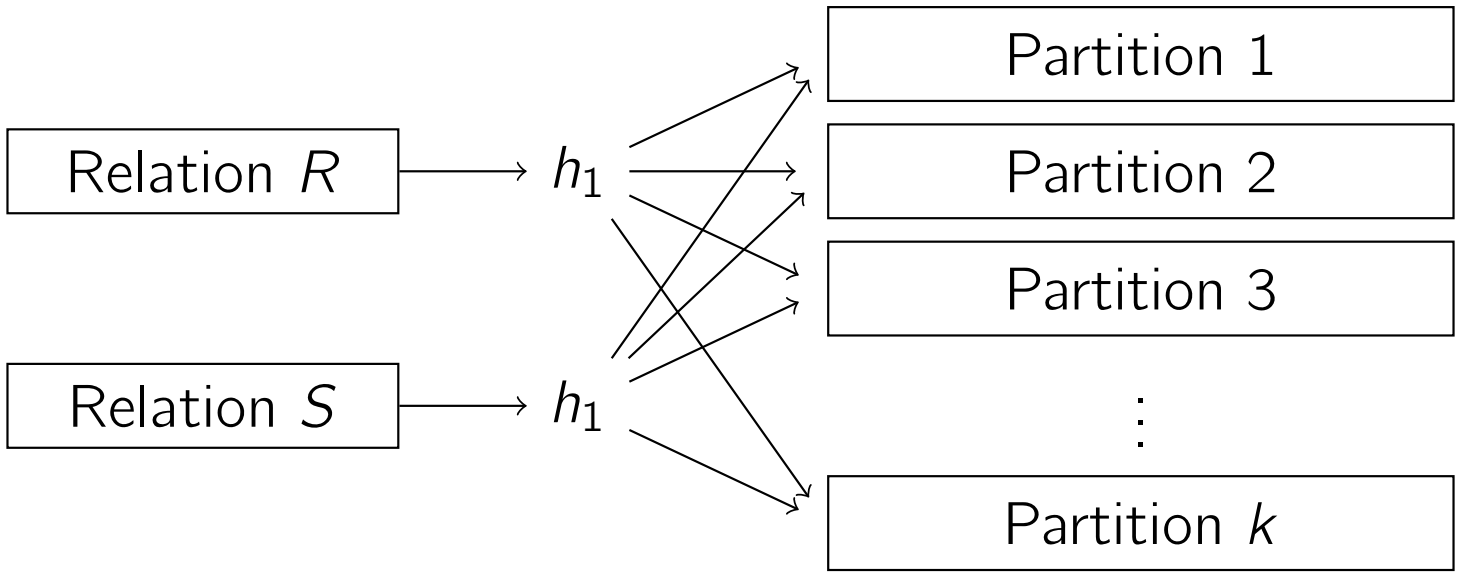
- Download application framework → `http://www.systems.ethz.ch/sites/default/files/file/dpmh_Fall2012/src-handout-02_tar.bz2`.
- Code builds two relations: R and S (random or zipf distribution)
- Simple *Nested Loops Join* is implemented
- Classical *Hash Join* is implemented
- Implement *Partitioned Hash Join*
- Improve partition phase → multiple passes
- Plot your results
- Profile your application and explain your results

Some Comments on the Application Framework

- `PartitionedHashJoin.c`
  - Turn off *Nested Loops Join* → `//#define NESTED_LOOPS_JOIN`
  - Use larger relations for *Hash Join* → `#define NUM_RELATION_R 8000000`
  - `NUM_BUCKETS`: How does *Hash Join* perform for different bucket sizes?
- `Relations.h`
  - `USE_ZIPF` : use *zipf* instead of *random* distribution
    `ZIPF_PARAM` : small value results in small skew, 1 is large
    `ZIPF_PERCENTAGE` : size of the value array compared to number of tuples (in [0,1])
  - `NPAD` : tuple padding (default = 0)

Partitioned Hash Join

Thus: **Partition** input into **cache-sized chunks**.



- Choose k ($\rightsquigarrow h_1$) such that single partitions **fit into cache**.
- Partition **both** input relations using the **same** hash function h_1.
- Only need to join **within** partitions then.

Partitioned Hash Join

This yields us a **partitioned hash join** algorithm:

```
1 Function: partitioned_hash_join(R, S)
  // Partition
2 partition R into R_1, ..., R_k ;
3 partition S into S_1, ..., S_k ;
  // Hash Join
4 foreach i ∈ 1, ..., k do
5     hash_join(R_i, S_i) ;
```

1. **Partition** both input relations.
2. Run "normal" **hash join on partitioned input**.
3. Hint: if partions are small, nested loops might perform better.

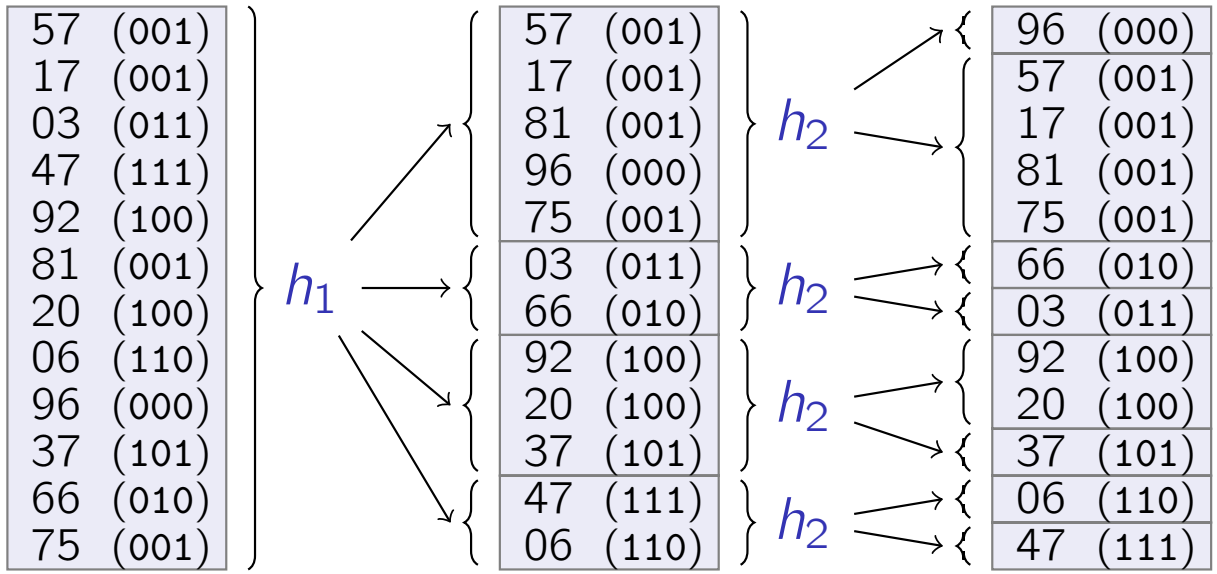
Task 1

- Implement `partitioned_hash_join(R,S)`
- Reuse *Classical Hash Join* code **or** write from scratch (e.g., in-place algorithm)
- Verify that your application is correct → result = number of matches
- *Entire* execution time is automatically displayed → measure *partition phase* and *join phase* separately
- When does *partition phase* become the bottle neck? Hint: use large relations, *e.g.*, 600 MiB.
- Use Oprofile for a better understanding
  - How does number of partitions affect cache miss rate?
  - Do the cache misses occur in the *partition* or in the *join phase*?
  - Can you figure out anything about TLB misses?

Improving the Partition Phase

Hardware determines the maximum number of partitions we can create **at the same time**.

Idea: Partition in **multiple passes**.



Radix-Cluster Algorithm:

- $h_1, \ldots, h_P$ use same hash function but look at different bits.

Task 2

- Find setting for your application, where *partition phase* dominates
- Implement `multipass_partitioned_hash_join(R,S)`
- Hint: Use Radix-Cluster Algorithm **or** other multi-pass partitioning scheme
- Can you improve overall performance?
- Use Oprofile for a better understanding
  - How are cache misses affected?
  - What about the TLB?
  - Hint: Oprofile events → `PAGE_WALKS` and `DTLB_MISSES`

Hand in your Results

- E-Mail to cagri.balkesen@inf.ethz.ch
- Subject = DPMH:assignment3_{your netzname}
- Body = Description of CPU you tested on, e.g., Xeon QuadCore L5520,(4x 2267MHz)
- Attach plots + raw data
- For this assignment, please attach your **source code**

Literature

This assignment is based on

- P. Boncz *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. `http://oai.cwi.nl/oai/asset/14832/14832A.pdf`, 1997.
- P. Mishra and M.H. Eich. *Join processing in relational databases*. ACM Computing Surveys, 1992.
- L.D. Shapiro. *Join processing in database systems with large main memories*. ACM Transactions on Database Systems (TODS), 1986.
- A. Shatdal, C. Kant, and J.F. Naughton. *Cache conscious algorithms for relational query processing*. VLDB, 1994.