

# Query Processing 4: Hash Join

Yufei Tao

<https://www.cse.cuhk.edu.hk/~taoyf>

This lecture will introduce the **hash join** algorithm for computing a natural join involving two relations.

## The Binary Join Problem

$R_1(X, Y)$ : A relation with attributes  $X$  and  $Y$ .

$R_2(X, Z)$ : A relation with attributes  $X$  and  $Z$ .

$B_1$  = the number of disk blocks that  $R_1$  occupies.

$B_2$  = the number of disk blocks that  $R_2$  occupies.

$M$  = the number of memory blocks (a.k.a., the buffer blocks).

- We assume that  $M \geq 7$  and  $M - 1$  is a multiple of 3.

**Goal:** Compute the join result  $R_1 \bowtie R_2$ .

Recall that the **block-based nested loop** (BNL) algorithm can compute the join result in  $B_1 + \lceil \frac{B_1}{M-1} \rceil B_2$  I/Os. In particular, if  $R_1$  fits in  $M - 1$  memory blocks, then the I/O cost of BNL is  $B_1 + B_2$ .

## Hashing

Let  $\mathbb{D}$  be the domain of the “join attribute”  $X$ . Set

$$U = (M - 1)/3$$

We are given a **hash function**  $H$ , which is a function mapping  $\mathbb{D}$  to the set of integers  $\{1, 2, \dots, U\}$ .

Given a tuple  $t_1$  of  $R_1$ , we refer to  $H(t_1.X)$  as the **hash value** of  $t_1$ .  
Given a tuple  $t_2$  of  $R_2$ , we refer to  $H(t_2.X)$  as the **hash value** of  $t_2$ .

We will carry out our discussion under the following assumption:

**Good hashing assumption:** For any  $h \in [1, U]$ , the tuples of  $R_1$  with hash value  $h$  can fit in  $U$  blocks.

No such requirements are placed on  $R_2$ .

## Buckets

Given a value  $h \in [1, U]$ , define

$$R_1(h) = \{\text{tuple } t \in R_1 \mid H(t_1.X) = h\}$$

$$R_2(h) = \{\text{tuple } t \in R_2 \mid H(t_2.X) = h\}$$

We will refer to  $R_1(h)$  and  $R_2(h)$  as **buckets**. Clearly, each relation has  $U$  buckets.

**Fact:**  $R_1 \bowtie R_2 = \bigcup_{h=1}^U R_1(h) \bowtie R_2(h).$

In other words, once we have obtained all buckets, we can focus on joining each  $R_1(h)$  with the **corresponding**  $R_2(h)$ .

## Hash Join

**Step 1:** Create the  $U$  buckets of  $R_1$  and  $R_2$ , respectively.

**Step 2:** For each  $h \in [1, U]$ , compute  $R_1(h) \bowtie R_2(h)$ .

We will discuss each step in turn.

## Hash Join: Step 1

We will explain how to create the  $U$  buckets of  $R_1$ .

Use one memory block as the **output buffer** for each bucket  $R_1(h)$ , where  $h \in [1, U]$ .

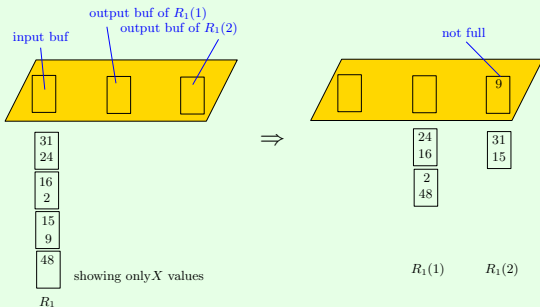
Use one memory block as the **input buffer** for reading  $R_1$ .

## Hash Join: Step 1

For each tuple  $t$  read from  $R_1$ , add it to the output buffer of  $R_1(H(t.X))$ .

When an output buffer — say for bucket  $R_1(h)$  of some  $h \in [1, U]$  — is **full**, append it to the file of  $R_1(h)$  on disk.

**Example:**  $U = 2$  and  $H(x) = 1 + (x \bmod 2)$ .

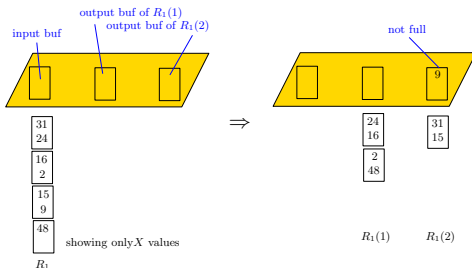




## Hash Join: Step 1

The I/O cost is bounded by  $2B_1$  because

- every block of  $R_1$  is read once;
- the number of full blocks flushed to the disk is at most  $B_1$ .



**Remark:** Each bucket can have **at most one non-full block**, and this block **must reside in memory**. We do not write those blocks to disk; otherwise, the I/O cost can increase to  $2B_1 + U$  in the worst case (**think:** why?).

## Hash Join: Step 1

Create the  $U$  buckets of  $R_2$  in the same way, while keeping the non-full bucket blocks of  $R_1$  in memory.

The I/O cost is bounded by  $2B_2$

At this moment, we have used  $2U$  memory blocks:

- $U$  blocks for  $R_1$  (each may be the non-full block of a bucket);
- $U$  blocks for  $R_2$  (each may be the non-full block of a bucket).

## Hash Join: Step 2

**Step 2:** For each  $h \in [1, U]$ , compute  $R_1(h) \bowtie R_2(h)$ .

Apply BNL to compute  $R_1(h) \bowtie R_2(h)$ . By the good hashing assumption,  $R_1(h)$  can be loaded into  $U$  memory blocks. Then, use one memory block to scan  $R_2(h)$ .

- In total, we use at most  $3U + 1 = M$  memory blocks.

The I/O cost is bounded by  $B_1 + B_2$  because every disk-resident block of the  $2U$  buckets is read exactly once.

Overall, the hash join algorithm performs  $3(B_1 + B_2)$  I/Os.

**Remark.** Recall that our analysis requires the good hashing assumption. **This assumption may not hold when data are “skewed”.** For example, if all the tuples of  $R_1(X, Y)$  share the same  $X$ -value, no hash function can satisfy the assumption as long as  $B_1 > U$ . In the presence of data skew, hash join can be prohibitively expensive.