TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Dr. Thomas Neumann

Exercises for *Foundations in Data Engineering*, WiSe 23/24

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)
http://db.in.tum.de/teaching/ws2324/foundationsde

Sheet Nr. 10

Exercise 1 Neojoin

Relation X and Y are distributed over three compute nodes as shown in Table 1 and Table 2. They shall be joined on $X.a = Y.b$. The compute nodes are connected by a star-shaped network. That means, there is one central switch which every node is connected to. All nodes have the same bandwidth towards the switch.

In preparation of executing a Neojoin, create one *partition to node* assignment that

1. minimizes network traffic
2. one that balances the amount of tuples on all nodes

Use $h(x) = x\ mod\ 3$ to partition the elements using relation X's attribute a and relation Y's b. Table 1 and Table 2 already contain the results of $h(x)$ for each tuple.

Then, create a minimal schedule for the balanced variant.

Table 1: Relation X, distributed over three nodes.

Node	a	$h(a) = a\ mod\ 3$
A	3	0
A	4	1
A	7	1
A	8	2
A	6	0
A	9	0
B	2	2
B	5	2
B	3	0
B	5	2
B	4	1
C	3	0
C	2	2
C	7	1
C	9	0

Table 2: Relation Y

Node	b	$h(b) = b\ mod\ 3$
A	3	0
B	4	1
C	2	2

Solution:

At first we determine how many tuples are in each partition. Therefore, we count for each node how many tuples belong to the different partitions. Table 3 contains the result.

Table 3: Number of tuples per partition and node.

Partition	**0**	**1**	**2**
Node A	4	2	1
Node B	1	2	3
Node C	2	1	2

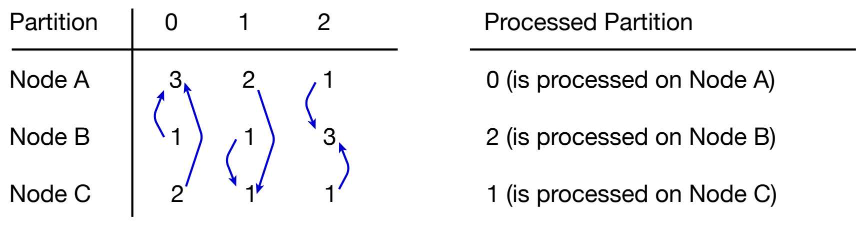
Then, we need to assign partitions to nodes for the two different strategies:

1. Minimize network traffic by not moving many tuples over the network. That means: Move every partition to the node with most tuples in the partition. Move *partition 0 to node A*, move *partition 1 to node A or node B*, move *partition 2 to node B*. This assignment is suboptimal, as node C does not have any partitions assigned. That means that it will not do any work during the join later on. Its resources are unused.
2. To assign a balanced amount of tuples on each node: move *partition 0 to node A*, *partition 1 to node C*, *partition 2 to node B*. This assigns one partition to every node, while creating the least traffic of all equal assignments.

For the schedule, each node moves 1 tuple around per time slot. Make sure that each node is receiving and sending in every time slot. Usually, empty slots are at the end of the schedule.

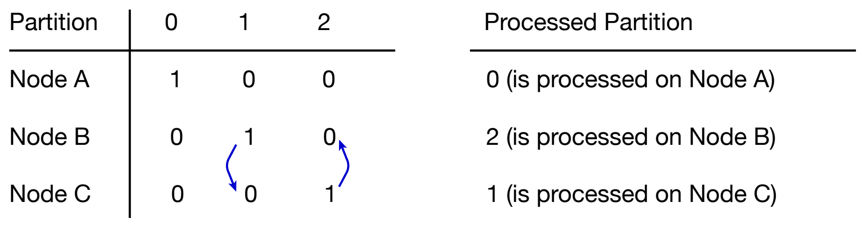
To create a schedule, first determine how many tuples have to be moved to other nodes. The tables below show for each node how many tuples of each partition it contains for Relation X and Relation Y. For example, node A contains for Relation X 3 tuples ($a = 3, 6, 9$) of partition 0, 2 tuples ($a = 4, 7$) of partition 1, 1 ($a = 8$) tuple of partition 2 and only 1 tuple ($b = 3$) of Relation Y, which is part of partition 0. The blue arrows indicate to which node the tuples need to be moved, e.g. node A's tuples of partition 1 have to be moved to node C.

Relation X:



Partition	0	1	2	Processed Partition
Node A	3	2	1	0 (is processed on Node A)
Node B	1	1	3	2 (is processed on Node B)
Node C	2	1	1	1 (is processed on Node C)

Relation Y:



Partition	0	1	2	Processed Partition
Node A	1	0	0	0 (is processed on Node A)
Node B	0	1	0	2 (is processed on Node B)
Node C	0	0	1	1 (is processed on Node C)

Eventually, we create the schedule, where each node sends and receives one tuple per time slot.

Schedule:

Time Slot	1	2	3	4
Node A	2	1	1	
B	1	0		1
C	0	2	0	2

Length: 4

Detailed description of the schedule:

Time slot 1:

- Node A sends tuple $a = 8$ to node B.
- Node B sends tuple $a = 4$ to node C.
- Node C sends tuple $a = 3$ to node A.

Time slot 2:

- Node A sends tuple $a = 4$ to node C.
- Node B sends tuple $a = 3$ to node A.
- Node C sends tuple $a = 2$ to node B.

Time slot 3:

- Node A sends tuple $a = 7$ to node C.
- Node B does not send a tuple.
- Node C sends tuple $a = 9$ to node A.

Time slot 4:

- Node A does not send a tuple.
- Node B sends tuple $b = 4$ to node C.
- Node C sends tuple $b = 2$ to node B.

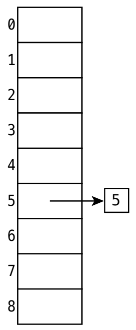
After 4 time slots all tuples reside on the correct node of their partition.

Exercise 2 Demonstrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \ mod \ 9$.

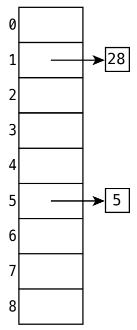
Solution:

Step	Value k	Hash value $h(k)$
1	5	5
2	28	1
3	19	1
4	15	6
5	20	2
6	33	6
7	12	3
8	17	8
9	10	1

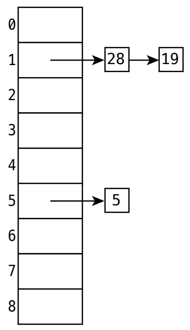
Step 1: Insert 5



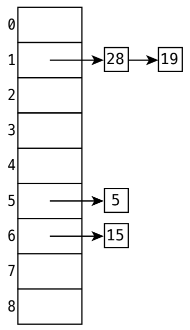
Step 2: Insert 28



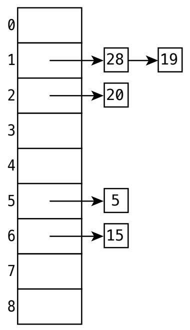
Step 3: Insert 19



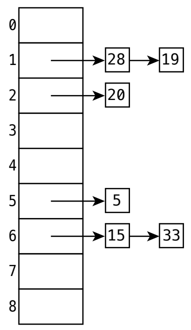
Step 4: Insert 15



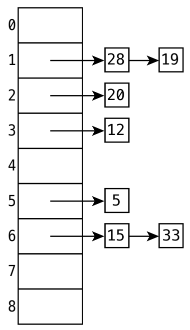
Step 5: Insert 20



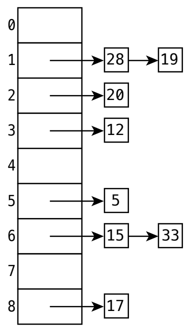
Step 6: Insert 33



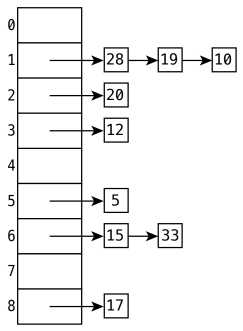
Step 7: Insert 12



Step 8: Insert 17



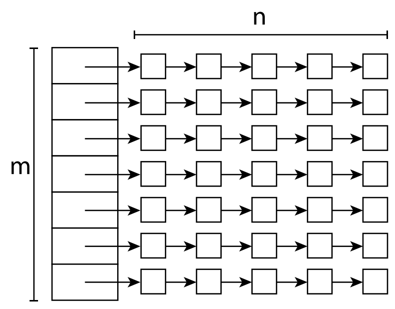
Step 9: Insert 10



Exercise 3 We have a hashtable with m buckets and insert $m * n$ elements. For this task we assume m is a constant and the universe of keys is U.

Show that if $|U| \geq n * m$, there is a subset of U of size $\geq n$ consisting of keys that all hash to the same bucket, so that the worst-case lookup time for hashing with chaining is $\Omega(n)$.

Solution:
Insert all $n * m$ elements into the hashtable with m slots.
In the worst case all elements fall into the same bucket, thus the worst-case lookup time is $\Omega(n)$. Even with a hash function which is able to balance perfectly, there must be at least one chain with n elements:
$\textit{elements per chain} = \frac{\#\textit{elements}}{\#\textit{chains}} = \frac{nm}{m} = n$
Thus, the worst-case lookup time is $\Omega(n)$.



Exercise 4 **Hashing and hash tables**

This exercise is about hash tables and properties of hash functions used for them. In order to get a feeling for implementation differences, we ask you to create your own implementations for this task.

1. Measure the performance of modulo with prime. Create some random integers and hash them. How many cycles does it take to compute one hash?
2. What is the performance when using modulo of power of two?
3. Using non-prime modulo as a hash function has a significant drawback. When all hashed values and the modulus have a factor in common, the domain of the hash is underutilized. In order to measure the impact of this, create a simple hashtable. Use hashing with chaining and implement the operations `insert` and `find`. Create some random integer data and insert it into the hashtable. Then, perform key lookups for all elements. How fast is the lookup when a prime is used? How fast is the lookup with powers of two? How fast is the lookup without maliciously constructed data?
4. Extra (not discussed in the tutorial): How fast is *Fibonacci Hashing*? Does it withstand the attack?

You can use the following `C++` function to measure cycles. Call the function before and after the code snippet you want to measure and calculate the difference, which is the number of cycles spent in the code snippet.

```
/// Read cycle counter for x86 (Intel & AMD)
uint64_t rdtsc() {
  unsigned int lo, hi;
  __asm__ __volatile__("rdtsc" : "=a"(lo), "=d"(hi));
  return ((uint64_t)hi << 32) | lo;
}
```

```
/// Read cycle counter for ARM
uint64_t rdtsc() {
  unsigned int val;
  __asm__ __volatile__("mrs %0, cntvct_el0" : "=r" (val));
  return (uint64_t) val;
}
```

Solution:
See folder Solution_hashtables.