CSCE 608 Database Systems Project 2
Part 2: Hash-Based Join

Texas A&M University
Luming Xu
UIN: 930001415

December 2020

Contents

1 Introduction

This project implements a two-pass hash-based algorithm to perform a binary operation of nature join. Two-pass hash-based algorithm works when the two relations are too large to completely fit into the main memory. The basic idea of the algorithm is to handle the memory constraint due to large relation size.

In this project, the constraint for two-pass hash-based algorithm is discussed and compared with theoretical limit by benchmark the hash function. Interface for virtual system components are also included. To verify that the implementation is correct, the generated relations and their join results are written to a local database PostgreSQL to compare with the algorithm result.

GitHub Repository: `https://github.com/Abalagu/HashBasedJoin`

1.1 Algorithm Overview

The algorithm includes two phases. In phase one, each relation loads into the main memory by disk block, then hash them to buckets according to their key column value. Since many operations only apply to those tuples with the same key value, which is equivalent to those tuples with the same hash value, performing hash operation serves as a grouping of the relation tuples, which greatly reduces the search space for binary operations such as nature join. Notice that tuples with the same hash value may not have the same key, but tuples with different value is sure to have different keys.

In phase two, two-pass hash-based algorithm utilizes sub-routine of one-pass algorithm. For one-pass algorithm performing binary operations, the smaller relation fully loads into the memory, and the larger relation loads block by block from disk, achieving a pair-wise operation while minimizing the total disk I/O. Two-pass hash-based algorithm simply converts the original problem to multiple one-pass problem, which reduces the constraint to that the smaller relation size(in unit of block) hashed into each bucket should not exceed the main memory size.

1.2 Derivation of Constraint

Suppose the selected hash function has n unique values, which corresponds to n buckets, and suppose memory has at most M blocks, and the relation size hashed to each bucket is $s_1, s_2, ..., s_n$.

1. Memory should hold one block for each bucket, $n \leq M$. The block assigned to each bucket within the memory serves as an intermediate collection that will be transferred to the disk when it is full.

2. For the total size of the relation hashed into each bucket, s_i, if it is the smaller relation, it should not exceed the main memory size, so that it does not violate the constraint of one-pass algorithm. For the larger relation, the total size within each bucket does no matter.

The inequalities summarized from the above discussion are:

- $s_i \leq M$, for i = 1, 2, ..., n
- $n \leq M$
- $sum(s_i) = B(R_{small})$ the block size of the smaller relation.

The memory constraint $M \geq \sqrt{B(R_{small})}$ in the slide page 98 is derived from an implicit assumption that each hash bucket has approximately the same total size. With this assumption, the extreme case is that each bucket holds M blocks of relation hashed into it, and there are M buckets, in total it would be $M \cdot M \geq B(R_{small}), M \geq \sqrt{B(R_{small})}$

However, such an even distribution of hashed results can only be achieved with a good hash function, along with an evenly distributed relation key value. The input relation is often not under control of the database system, and if the distribution of hash values is a bit too uneven for a certain buckets, it may violate the one-pass algorithm constraint, while it appears doable with two-pass based algorithm. Also, one cannot tell whether a hash function is good enough prior to the inspection of the relation key value distribution. Therefore, the hash function needs to be benchmarked to ensure that the selected function yields suitable results for the given relation. During the implementation of this algorithm, the above discussed problem arises, that one cannot pre-allocate blocks on disk with size `ceil(B(R)/n)` for each bucket, otherwise some buckets will run out faster than the others.

2 Data Generation

A routine generates tuples for the two relations R(A, B) and S(B, C). In the project requirement document, it mentions that attribute A and C can be of any type. For the ease of debugging, `A = str(B + randint(1, 100)), C = str(B)`. For example, valid tuples for R(A, B) are `['A12345', 12345], ['A23903', 23827]`; valid tuples for S(B, C) are `[18272, 'C18272'], [40504, 'C40504']`.

3 Virtual System Design

3.1 Virtual Block

Virtual block is the assimilated unit of disk storage block. The maximum storage capacity within a block is 8 tuples.

```
class Block:
    def read(self):
        """return content within the block"""
        pass

    def write(self, data):
        """overwrite content within the block.  """
```

```
        pass

    def verify(self):
        """check that the number of data within is under the
    constraint"""
        pass

    def clear(self):
        """clear the content within the block"""
        pass
```

Memory block inherits all the disk block functionalities, with an extra method as extending the content within a block, which is frequently used when hashing tuples to a memory block.

```
class MemoryBlock(Block):
    def extend(self, data: List[Tuple]) -> None:
        """extend the content within the memory block"""
        pass
```

3.2 Virtual Disk

Virtual disk organizes a collection of virtual disk blocks, and provides methods to read and write disk blocks.

```
class VirtualDisk:
    def append(self, data: List[Tuple]) -> range:
        """append the given data at the end of the last used block.
      for large amount of data, slide into pieces
        suitable for each unit block.  return the disk index range
    of the new blocks. """
        pass

    def allocate(self, size: int) -> range:
        """allocate empty blocks of given number after the last
    used block.  return block index range."""
        pass

    def get_block(self, idx: int) -> Block:
        """return a block at the given index."""
        pass

    def read(self, idx: int) -> List[Tuple]:
        """get block at the given index, read and return its
    content"""
        pass

    def write(self, data: List[Tuple], idx: int) -> None:
        """overwrites the disk data at the given index"""
        pass
```

3.3 Virtual Memory

Virtual memory supports similar methods as the virtual disk, which organizes the memory blocks.

```python
class VirtualMemory:
    def get_block(self, idx: int) -> MemoryBlock:
        pass

    def is_empty(self, idx: int)->bool:
        """check if the memory block is empty.  """
        pass

    def read(self, idx: int) -> List[Tuple]:
        pass

    def write(self, data: List[Tuple], idx: int) -> None:
        pass

    def extend(self, data: List[Tuple], idx: int) -> None:
        pass

    def clear(self, idx: int) -> None:
        """notice that writing to a memory block with existing content may override data.
        using block.clear() after the block usage is encouraged.
    """
        pass
```

3.4 Virtual Table

A virtual table class is used to store exchange information relevant to a table.

```python
class Table:
    def __init__(self, name: str, key_idx: int, disk_range: Union[range, List], size: int, definition: str):
        """ Object to exchange information about a virtual table

        @param name: table name
        @param key_idx: key column index of a tuple
        @param disk_range: supports list and range.  list if stored in block fragments, range if stored contiguously.
        @param size: number of tuples within a relation
        @param definition: store the data definition language for reference if provided
        """
        pass
```

3.5 Virtual Database

Virtual database is the core of this two-pass hash-based algorithm. It implements one-pass hash-based algorithm, and is called by the two-pass algorithm.

```python
class VirtualDatabase:
    def __init__(self, virtual_memory: VirtualMemory, virtual_disk: VirtualDisk):
        """mounts memory and disk for relation operations"""
        pass

```

```
    def add_table(self, table_name: str, key_idx: int, disk_range: range, num_tuples: int) -> None:
        pass

    def disk_to_memory(self, disk_idx, memory_idx) -> None:
        """transfer content within a disk block to a memory block. exception is raised if the memory is not empty
        to encourage explicit clear of memory block after usage.
        """
        pass

    def memory_to_disk(self, memory_idx, disk_idx) -> None:
        """transfer content within a memory block to a disk block"""
        pass

    def hash_relation(self, table: Table) -> Dict[int, List]:
        """hash relation into buckets stored in the disk, then return the block range"""
        pass

    def one_pass_nature_join(self, table_1: Table, table_2: Table) -> List[Tuple]:
        """perform one-pass hash-based algorithm in memory nature join.
        table sizes are checked before operation to ensure viability.
        output is not stored back to disk and shall be handled by the upper layer."""
        pass

    def nature_join(self, table_1: Table, table_2: Table) -> List[Tuple]:
        """hash-based two-pass algorithm for nature join
        1. hash both relations into buckets
        2. load the small relation content into memory from each bucket
        3. perform one-pass nature join with large relation blocks loaded from disk iteratively"""
        pass
```

4 Hash Relation

A hash function is "good" for a relation if each resulting bucket size does not violate the two-pass hash-based algorithm constraint, that the total size of each bucket for smaller relation can fit into the memory. In this sense, a hash function needs to be verified, and one cannot tell whether a given hash function is "good" by its structure in a general sense.

4.1 Hash Function

Integer modulo function is picked to be the hash function. The reasons are:

1. the key of the relations are integer numbers.

2. the number of bucket can be easily controlled by the divisor.

As the memory constraint is 15 blocks, it leaves one block for temporary storage of reading block to disk, and the rest 14 blocks as hash buckets. Therefore, the modulo divisor is set to 14.

4.2 Verify Hash Function

The constraints are listed as follows:

- virtual main memory: 15 blocks
- virtual disk: unlimited
- relation S(B, C): 5000 tuples
- range of B: [10000, 50000]
- maximum number of tuples per block: 8
- join based operation can only be performed when the tuples are in the virtual main memory.

For experiment 1, R(A,B) is of 1000 tuples, which corresponds to 125 blocks. For experiment 2, R(A,B) is of 1200 tuples, which corresponds to 150 blocks. For both experiment 1 and 2, S(B,C) is of 5000 tuples, which corresponds to 625 blocks. It is obvious that in both cases, the smaller relation is R(A,B), which is the concern of whether it conforms to the one-pass hash-based algorithm constraint, which is $M \geq \sqrt{B(R)}$. From constraint rule, the valid bucket sizes are as follows:

- For experiment 1, $M \geq \sqrt{125} = 11.18, M = 12, 13, ...$
- For experiment 2, $M \geq \sqrt{150} = 12.24, M = 13, 14, ...$

4.2.1 Benchmark

I wrote a benchmark function to test whether these bucket sizes would be sufficient to hold content hashed into each bucket. The main idea is to hash the relation by a given bucket size n, count the maximum number of tuples among all the buckets from 0 to n-1, `ceil(num_tuples)/8)` would be the maximum number of blocks required to hold the relation. Also, the trial is repeated for several times to see if it generalizes well.

The benchmark result log is as follows:

```
1 experiment 1, bucket size: 12 blocks.  max block usage: 15, met:
      violated = 18:82
2 experiment 1, bucket size: 13 blocks.  max block usage: 14, met:
      violated = 97:3
3 experiment 1, bucket size: 14 blocks.  max block usage: 13, met:
      violated = 100:0
```

```
experiment 2, bucket size: 12 blocks.  max block usage: 17, met:
    violated = 0:100
experiment 2, bucket size: 13 blocks.  max block usage: 16, met:
    violated = 21:79
experiment 2, bucket size: 14 blocks.  max block usage: 15, met:
    violated = 96:4
```

For example, for experiment 1, if using 12 buckets, the maximum number of blocks used is 15 blocks for one bucket, among the 100 trials. Also, the constraint is met 18/100 times. It is easy to see that using 14 blocks may be safe for experiment 1, but it could cause some problem for experiment 2, although during the testing it rarely happens.

For the actual detail of the benchmark, see file `hash_test.py`

4.3 Hash Bucket Block Disk Storage

When performing hash in memory, if a memory bucket/block is full, then it needs to write back to disk. This comes with a design choice of how the relation should be stored on disk. The two options are either to store in a contiguous block range, or in block fragments.

4.3.1 Contiguous Storage

A relation may be contiguously stored within a disk range when it first enters the database. One may also think that after hashing a relation, the partial relation with a same hash key should be stored contiguously. If all the bucket blocks are to be stored continuously, then one needs to allocate a block range for each bucket. However, this is not easy to achieve. When the temporary hash buckets stored within memory is filled with tuples, it should transfer back to disk. The order that the bucket is filled is not under the control of hash algorithm.

One may optimistically estimate that due to evenness of hash function behavior, the number of disk blocks that each bucket holds is $B(R)/n$, where n is the number of buckets. However, even for a relation generated purely from randomness, as is the case in the experiment, it cannot be avoided that certain buckets hold slightly more than other buckets. In this case, the preallocation of disk range to each bucket will fail.

4.3.2 Fragment Storage

To solve the above problem, the resulting blocks from hashing a relation are stored in fragments. In this case, the surrounding blocks may not belong to the same bucket, and are in order of time that the buckets are fills. Therefore, there needs to be a memo on the index list of each bucket within the disk, such as:

```
bucket_disk_idx = {
    0:[0,2,4,8],
    1:[1,3,5,7],
    # ...
}
```

When hashing the smaller relation, if one of the bucket index list exceeds memory block size, then it indicates that the chosen hash function is not good enough. This is used to detect whether a hash function is suitable for the given relation.

4.3.3 Comparison

After completing the hash of a relation, the resulting fragmented disk blocks may be swapped to make it that each bucket content stores contiguously, but it is not implemented in this program. One may argue that the disk block retrieval becomes faster when blocks are near. Still, since block retains a minimal read/write unit, the performance is not very much penalized, and the author sees no obvious performance benefit in rearranging the blocks, which also takes disk I/O as overhead.

5 Join Algorithm

The program first implements one-pass hash-based algorithm, then it is called by the two-pass hash-based algorithm. The total number of disk IO can be estimated as follows. Suppose the database is to perform nature join on a small relation **S** and a large relation **L**. In phase one, it loads each relation block by block, then hash to disk, resulting in around B(**S**) and B(**L**) of blocks. Notice that it could result in slightly more blocks than the original relation, since some blocks may not be completely filled.

```
def one_pass_nature_join(relation_s, relation_l):
    block_s <- load all the blocks from small relation

    for block in relation_l:
        block_l <- load a disk block from relation l
        for tuple_l in block_l:
            for tuple_s in block_s:
                if tuple_l and tuple_s have the same key:
                    perform nature join, direct to output
```

```
def two_pass_nature_join(relation_s, relation_l):
    hash relation_s to buckets
    hash relation_l to buckets

    for each bucket index:
        block_s <- load all blocks from relation_s with the given
    index

        for each block in relation_l with the given index:
            block_l <- load one block
            ret = one_pass_nature_join(block_s, block_l)
```

It is easy to estimate the total number of disk I/O required to complete two pass operation. In phase one, each disk block is read once for hashing, write once back to disk, in total of $2 \times (B(S) + B(L))$ In phase two, for each bucket, fully loads blocks from smaller relation, while load one block at a time from

larger relation. This takes in total of $B(S) + B(L)$ read. The output from the operation is not counted. Therefore, in total it takes $3 \times (B(S) + B(L))$. For the first experiment, it is estimated to have total disk I/O of $3 \times (125 + 500) = 1875$ blocks, with 1250 read and 625 write. For the second experiment, it is estimated to have total disk I/O of $3 \times (150 + 500) = 1950$ blocks, with 1300 read and 650 write.

6 Verify

The experiment function prints out the nature join results as is required in the project document. However, it may not be easy to see whether the results are correct and complete. To verify more easily, the program connects to a local PostgreSQL database to create an actual relation for each table, and compare that it yields the correct result.

6.1 Test Log

Below is the test output log for the first experiment.

```
FIRST EXPERIMENT:
AFTER INIT:
blocks used: 0, read count: 0, write count: 0
AFTER CREATION:
blocks used: 750, read count: 0, write count: 750
AFTER HASHING SMALL RELATION:
blocks used: 882, read count: 125, write count: 882
AFTER HASHING LARGE RELATION:
blocks used: 1514, read count: 750, write count: 1514
AFTER NATURE JOIN:
blocks used: 1514, read count: 1514, write count: 1514
TABLES INIT.
R(A,B), S(B,C), AND JOIN RESULTS WRITTEN TO DATABASE
```

Below is the test output log for the second experiment.

```
SECOND EXPERIMENT:
AFTER INIT:
blocks used: 0, read count: 0, write count: 0
AFTER CREATION:
blocks used: 775, read count: 0, write count: 775
AFTER HASHING SMALL RELATION:
blocks used: 930, read count: 150, write count: 930
AFTER HASHING LARGE RELATION:
blocks used: 1560, read count: 775, write count: 1560
AFTER NATURE JOIN
blocks used: 1560, read count: 1560, write count: 1560
tables init.
R(A,B), S(B,C), AND JOIN RESULTS WRITTEN TO DATABASE
```

From the above printed out disk usage stat, the disk I/O count is within the range of estimation. Because the table generation it writes to the disk, write count is more than estimated by the block size of each table. Also, the hashed

relation takes more blocks of storage because some tail blocks of a bucket are not full.

Note that the nature join output is too long to be attached in the report. It can be verified by comparing the console output result and the database query result.