W4111 - Introduction to Databases Section 02, Fall 2019

UNI – PKV2103

Overview

This assignment requires written answers to questions. You may have to copy and paste some SQL statements or scripts into the answer document you submit. You may also have to insert diagrams.

There are 20 questions worth 5 points each. A homework assignment contributes 10 points to your final score. Dividing your score on this assignment by 10 yields the points credited to your final score.

Question 1

Question: Suppose you have data that should not be lost on disk failure, and the application is write-intensive. How would you store the data? (Note: Your answer should consider differences between primarily sequential writes and primarily random writes).

Answer: Redundant Array of Independent Disks (RAID) can be used to handle disk failures. There are several RAID options, each with different cost and performance implications.

For write intensive data, when it is primarily sequential write, RAID 1 and RAID 5 will both perform well but with less storage cost for RAID 5. RAID 5 however has significant overhead for random writes, since a single random block write requires 2 block reads (to get the old values of block and parity block) and 2 block writes to write these blocks back. RAID level 1 is therefore the RAID level of choice for many applications with primarily random writes requirement.

RAID level 6 offers better reliability than level 1 or 5, since it can tolerate two disk failures without losing data. In terms of performance during normal operation, it is similar to RAID level 5, but it has a higher storage cost than RAID level 5. RAID level 6 is used in applications where data safety is very important.

Question 2

Question: Both database management systems (DBMS) and operating systems (OS) provide access to files and implement optimizations for file data access. A DBMS provides significantly better performance for data in databases. How does the DBMS do this? What information does it use? How does it optimize access? Provide some examples.

Answer: DBMS provides better performance for data in databases by storing the data in a structured manner in tables (or references) and providing appropriate queries to access the required data.

A DBMS can predict query access patterns, with some errors. DBMS can gather statistical information about data and historical queries to predict future access patterns for tables, records and blocks. It can predict future record block access during scan for Select and Join processing.

Here are some of the techniques employed by DBMS to optimize access:

- Optimize placement of blocks for sequential data. Sequentially accessed blocks are placed in order in the same cylinder, and using adjacent cylinders if the number of the blocks exceed cylinder capacity.
- Prefetch data into buffer by predicting future block access to implement a Select and Join.
- Supports column-oriented storage, also called a columnar storage. Each attribute of a relation is stored separately in compressed format, with values of the attribute from successive tuples stored at successive positions in the file. When a data analysis query needs to access only a few attributes of a relation with a large number of attributes, the remaining attributes need not be fetched from disk into memory. This reduces the I/O significantly.
- LRU is an acceptable replacement scheme in operating systems. However, a database system is able to predict the pattern of future references more accurately than an operating system. A user request to the database system involves several steps. The database system is often able to determine in advance which blocks will be needed by looking at each of the steps required to perform the user-requested operation. Thus, unlike operating systems, which must rely on the past to predict the future, database systems will have information regarding at least the short-term future.
- The optimizer plugs the information on Statistics into a series of complex formulas that it uses as it builds optimized access paths. Statistics used by the optimizer include information about the current status of the tables, indexes, columns, and table spaces (including partitioning information) that need to be accessed. Examples of typical statistics stored in the system catalog include table (space) size, clustering details, number of rows, number of distinct values for columns, index levels, and so on.
- In addition to the system catalog statistics, the optimizer will consider other system information such as the CPU being used, DDL options, and the amount of memory available. This allows the optimizer to estimate the number of rows that qualify for each predicate, and then use the proper algorithm to most efficiently access the required data.

Examples:

- Consider a user wants to access data from a directory for name, contact information and address of the person. For this, the user needs to use some queries to access the information via DBMS while in case of OS, the user needs to find the folder in which the data is stored and then search for it. The DBMS provides an efficient way to access the information by organizing the directory information efficiently using indexes and storing that information in a catalog that helps the optimizer to retrieve the data faster.
- Consider the data analysis and data mining use cases. These are rising fields that needs access to large amount of data. The usage of columnar format on such data analytics reducing the I/O significantly and provides quick, correct and consistent results as compared to that of accessing files via OS.

Question 3

Question: Briefly explain CHS addressing and LBA for disks. Which approach is more common and why? Is it possible to convert a CHS address to an LBA. If yes, how?

Answer: CHS (Cylinder-Head-Sector addressing) and LBA (Logical Block Addressing) are two different ways of accessing a location within a drive.

CHS is the most low-level method of determining where to read or write from the drive. It is a 3D-coordinate system made out of a vertical coordinate head, a horizontal (or radial) coordinate cylinder, and an angular coordinate sector. Head selects a circular surface: a platter in the disk (and one of its two sides). Cylinder is a cylindrical intersection through the stack of platters in a disk, centered around the disk's spindle. Combined together, cylinder and head intersect to a circular line, or more precisely: a circular strip of physical data blocks called track. Sector finally selects which data block in this track is to be addressed, and can be viewed as a sort of angular component. The sector is the smallest addressable unit, and was traditionally fixed at 512 bytes.

Logical block addressing (LBA) is a common scheme used for specifying the location of blocks of data stored on computer storage devices, generally secondary storage systems such as hard disk drives. LBA is a particularly simple linear addressing scheme; blocks are located by an integer index, with the first block being LBA 0, the second LBA 1, and so on.

CHS is the older scheme while LBA is currently used in all modern operating systems. For CHS, the possible values for the cylinder is 1023, while heads can be 255 maximum, and sector can only go up to 63, meaning you can have at most 1024 cylinders x 255 heads x 64 sectors x 512 bytes mapped in traditional CHS format, giving a grand total of under 8 GiB! So LBA was introduced with a 32-bit limit giving 2^32 x 512 bytes or 2 TiB limit on disk size. Newer, better options have been introduced like the GPT partitioning scheme which extends LBA to 64 bits

LBA numbering starts with the first cylinder, first head, and track's first sector. Once the track is exhausted, numbering continues to the second head, while staying inside the first cylinder. Once all heads inside the first cylinder are exhausted, numbering continues from the second cylinder, etc. CHS tuples can be mapped to LBA address with the below formula:

LBA = (((Cylinder * HPC) + Head) * SPT) + Sector – 1

Where,
HPC = the number of heads (per cylinder, or the number of read-write heads).
SPT = the number of sectors (per track/side)
Cylinder, Head and Sector are the Cylinder number, Head number and Sector number

To reverse this algorithm, that is, to find out the CHS address equal to an LBA address, the below formulae can be used.

1. Sector = (LBA mod SPT) + 1
2. Head = (LBA / SPT) mod HPC
3. Cylinder = (LBA / SPT) / HPC

Question 4

Question: Explain why the allocation of records to blocks affects database-system performance significantly.

Answer: Typically disk accesses are bottlenecks in database system's performance. And the data transfer between main memory and disk storage occurs in block units. If we allocate related records to blocks, we can often retrieve most or all of the requested records by a query with one disk access.

Since this allocation strategy reduces the number of disk accesses for a given operation, it significantly improves performance. And when related records are allocated to different blocks, then as many disk accesses are required impacting performance negatively.

Question 5

Question: Give benefits and disadvantages of variable length record management versus fixed length record management

Answer: We can organize a file logically as a sequence of records mapped onto disk blocks. One approach to mapping the database to files is to use several files, and to store records of only one *fixed length* in any given file. An alternative is to structure files so that they can accommodate *variable lengths* for records. The slotted page method is widely used to handle variable-length records within a disk block.

Benefits and Disadvantages of variable length record management versus fixed length:

Benefits:

- **Less Storage space is required** in variable-length record format. In fixed-length records, the records may hold additional redundant storage space to preserve the file size.
- **Take less time to read**. Records are more tightly packed making data search quicker in variable-length format.
- **Data is less likely to be truncated**. In fixed-length records, the additional data sent will be truncated.
- **New fields can be added easily**. Unlike fixed-length records, variable-length format can accommodate new fields in the file by adding the offset value for them in each record.
- **Can support records with dynamically changing column sizes**. As there is no pre-defined field size in variable-length records, new system requirements to increase/decrease the length can be accommodated.

Disadvantages:

- **Separating fields is a more complex task**. In variable length records, the start and end of each field is in a different place of each record. When reading a record, it must be split into the information stored under each field. Fixed-length record would go by the offset defined the record format.
- **File sizes are hard to estimate**. In a variable-record format, it's difficult to estimate database size because of the variable nature of the records. Fixed length formats have a predefined size for the records.

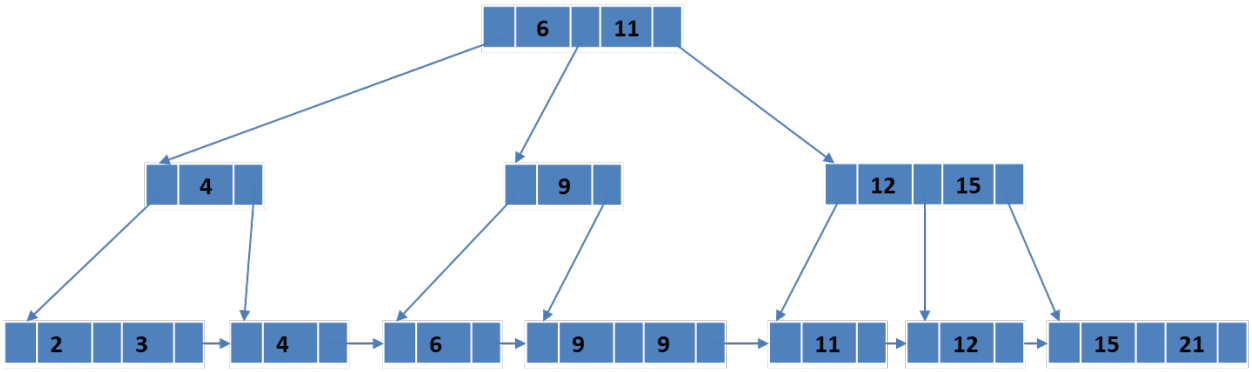
- **Records cannot be updated in situ** on variable-length records. In fixed length format, you would be able to overwrite the field without erasing anything else.

Question 6

Question: Build and draw a B+ tree after inserting the following values. Assume the maximum degree of the B+ tree is 3.

Values: 3, 11, 12, 9, 4, 6, 21, 9, 15, 2

Answer:



Given that the B+ tree degree of 3 means that any node can have a maximum of 2 keys. After inserting every element, check overflow. If overflow exists, push the middle element to parent node and place the same element in the right side children. The final tree after completion of all inserts is shown above.

Question 7

Question: Perform the same insertions in the same order for a hash index. Assume that:

1. The size of the hash table is 13.
2. The hash function is simple modulo.
3. The size of a bucket is one entry.
4. The size of each bucket is one value.
5. The index algorithm uses linear probing to resolve conflicts/duplicates.

Answer:

Hash Position	Value
0	
1	
2	15
3	3
4	4
5	2
6	6
7	
8	21
9	9
10	9
11	11
12	12

Shown above is the final hash table. Below are the steps taken during insertion of the data:

Hash position = 3 % 13 = 3
Attempting to insert 3 at position 3.
Inserting 3 at position 3.

Hash position = 11 % 13 = 11
Attempting to insert 11 at position 11.
Inserting 11 at position 11.

Hash position = 12 % 13 = 12
Attempting to insert 12 at position 12.
Inserting 12 at position 12.

Hash position = 9 % 13 = 9
Attempting to insert 9 at position 9.
Inserting 9 at position 9.

Hash position = 4 % 13 = 4
Attempting to insert 4 at position 4.
Inserting 4 at position 4.

Hash position = 6 % 13 = 6
Attempting to insert 6 at position 6.
Inserting 6 at position 6.

Hash position = 21 % 13 = 8
Attempting to insert 21 at position 8.
Inserting 21 at position 8.

Hash position = 9 % 13 = 9
Attempting to insert 9 at position 9.
Attempting to insert 9 at position 10.
Inserting 9 at position 10.

Hash position = 15 % 13 = 2
Attempting to insert 15 at position 2.
Inserting 15 at position 2.

Hash position = 2 % 13 = 2
Attempting to insert 2 at position 2.
Attempting to insert 2 at position 3.
Attempting to insert 2 at position 4.
Attempting to insert 2 at position 5.
Inserting 2 at position 5.

*source - http://iswsa.acm.org/mphf/openDSAPerfectHashAnimation/perfectHashAV.html

Question 8

Question: When is it preferable to use a dense index rather than a sparse index? Explain your answer.

Answer: Index schemes on databases are used for faster data retrieval. When a dense index is created, there exists an index entry for each search key value in the file. And for Sparse index, an index entry exists only for a few search key values. Sparse index relies on the concept that records are sorted in the same order as the index field.

Dense indexes are faster in general but Sparse indexes require less space and impose less maintenance for insertions and deletions.

Dense index is preferred over sparse index when the file is not sorted on the indexed field (secondary index is one such example) or when the index file is small compared to the size of memory.

Question 9

Question: Since indexes improve search/lookup performance, why not create an index on very combination of columns?

Answer:

Please find below the key properties of indices:

- Every index requires additional CPU time and disk I/O overhead during inserts and deletions.
- Each index requires additional storage space in RAM and disk.
- Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes).

As described above, there is an overhead in maintaining indices. We should only define indices that support our application requirements. Too many indices, as in the very combination of columns, will slow down the writes more than the gains from the reads.

In addition, for queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them. Therefore, database performance is improved less by adding indices when many indices already exist.

Question 10

Question: Consider the table below. Add indexes that you think are appropriate for the table and explain your choices. You may use MySQL Workbench to add the indexes. Paste the resulting create statement in the answer section.

Choosing indexes is not possible without understanding use cases/access patterns. Define five use cases and the index you define to support the use case. See the answer section for an example.

```
CREATE TABLE IF NOT EXISTS `customers` (
`id` INT(11) NOT NULL AUTO_INCREMENT,
`company` VARCHAR(50) NULL DEFAULT NULL,
`last_name` VARCHAR(50) NULL DEFAULT NULL,
`first_name` VARCHAR(50) NULL DEFAULT NULL,
`email_address` VARCHAR(50) NULL DEFAULT NULL,
`job_title` VARCHAR(50) NULL DEFAULT NULL,
`business_phone` VARCHAR(25) NULL DEFAULT NULL,
`home_phone` VARCHAR(25) NULL DEFAULT NULL,
`mobile_phone` VARCHAR(25) NULL DEFAULT NULL,
`fax_number` VARCHAR(25) NULL DEFAULT NULL,
`address` LONGTEXT NULL DEFAULT NULL,
`city` VARCHAR(50) NULL DEFAULT NULL,
`state_province` VARCHAR(50) NULL DEFAULT NULL,
`zip_postal_code` VARCHAR(15) NULL DEFAULT NULL,
`country_region` VARCHAR(50) NULL DEFAULT NULL)
```

Answer:

Use Case 1: A user wants to be able find a customer(s) by country_region; country_region and

state_province; country_region, state_province, city; just by city.

```
ALTER TABLE `W4111Midterm`.`customers`
ADD INDEX `index1` (`country_region` ASC, `state_province` ASC,`city` ASC) VISIBLE;

ALTER TABLE `W4111Midterm`.`customers`
ADD INDEX `index2` (`city` ASC) VISIBLE;
;
```

Use Case 2: A user wants to ensure email address are unique and would like to search the customers based on email address.

```
ALTER TABLE `W4111Midterm`.`customers`
ADD UNIQUE INDEX `index3` (`email_address` ASC) VISIBLE;
;
```

Use Case 3: A user wants to be able to find a customer(s) working in a given company with job title as 'Manager'

```
ALTER TABLE `W4111Midterm`.`customers`
ADD INDEX `index4` (`company` ASC, `job_title` ASC) VISIBLE;
;
```

Use Case 4: A user wants to find a customer(s) with the given first name and last name.

```
ALTER TABLE `W4111Midterm`.`customers`
ADD INDEX `index5` (`first_name` ASC, `last_name` ASC) VISIBLE;
;
```

Use Case 5: A user wants to be able to find a customer(s) by zip_postal_code; zip_postal_code, company

```
ALTER TABLE `W4111Midterm`.`customers`
ADD INDEX `index6` (`zip_postal_code` ASC, `company` ASC) VISIBLE;
;
```

Question 11

Question: Assume that:

1. The query processing engine can use four blocks in the buffer pool to hold disk blocks.
2. The records are fixed size, and each block contains 3 records.
3. There are two relations are R and S. Their formats on disk are given.

Explain how a partition hash join would perform a natural join. You should illustrate your explanation using diagrams of the form below for various steps in the processing:

The notation (n,X) means the record in file/relation X with value n for the key. Ti is a temporary file/relation that is created during the processing. You will likely have to create more than one

temporary files.

Answer:

The hash partition algorithm avoids rescanning the entire S relation by first partitioning both R and S via a hash function, and writing these partitions out to disk. The algorithm then loads pairs of partitions into memory, builds a hash table for the smaller partitioned relation, and probes the other relation for matches with the current hash table. Because the partitions were formed by hashing on the join key, it must be the case that any join output tuples must belong to the same partition.

It is possible that one or more of the partitions still does not fit into the available memory, in which case the algorithm is recursively applied: an additional orthogonal hash function is chosen to hash the large partition into sub-partitions, which are then processed as before. Since this is expensive, the algorithm tries to reduce the chance that it will occur by forming the smallest partitions possible during the initial partitioning phase.

Here are two high level phases in the algorithm:

- Partitioning phase
  - Apply a hash function h(x) to the join attributes of the tuples in both R and S. Assume 'n' buckets.
  - According to the hash value, each tuple is put into a corresponding bucket. Write these buckets to disk as separate files.

- Joining Phase:
  - Use the basic hash-join algorithm
  - Get one partition of R and the corresponding partition of S and apply the basic hash algorithm using a different hash function

Below is the algorithm:

```
/* h[1..n]: range of hash function; R[1..n] and S[1..n] are buckets */
for ( each tuple r in R){
  apply hash function to the join attributes of r;
  put r into the appropriate bucket R[i]
}

for (each tuple s in S){
  apply hash function to the join attributes of s;
  put r into the appropriate bucket S[i]
}

for (i=1; i <= n; i++){
    build the hash table for R[i]; /* using a different hash function h2*/
    for (each tuple s in S[i]){
      apply the hash function h2 to the join attributes of S;
      use s to probe the hash table;
      output any matches to the result relation;}
}
```

Illustration of the Partition has on the data provided:

Step 0 (Partitioning Phase): Load the R relation into buffer to partition it into 3 buckets using mod 3. Load the buffer with mod3 as 0.

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(3,T1), (6,T1), (9,T1)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
(1,R),(2,R),(3,R)		**T1**			
(4,R),(5,R),(6,R)		**T2**			
(7,R),(8,R),(9,R)		**O/P**			

Step 1 (Partitioning Phase): Buffer values written to disk partition. Mod3 as 1 loaded to output buffer

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(1,T1),(4,T1),(7,T1)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
(1,R),(2,R),(3,R)		**T1**	(3,T1), (6,T1), (9,T1)		
(4,R),(5,R),(6,R)		**T2**			
(7,R),(8,R),(9,R)		**O/P**			

Step 2 (Partitioning Phase): Buffer values written to disk partition. Mod3 as 2 loaded to output buffer

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(2,T1),(5,T1),(8,T1)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
(1,R),(2,R),(3,R)		**T1**	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	
(4,R),(5,R),(6,R)		**T2**			
(7,R),(8,R),(9,R)		**O/P**			

Step 3 (Partitioning Phase): Buffer values written to disk partition.
Load the relation S and the corresponding Mod3 as 0 paritition to output buffer.

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(21,T2),(3,T2),(27,T2)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
(11,S),(4,S),(21,S)		**T1**	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(3,S),(31,S),(13,S)		**T2**			
(5,S),(27,S),(23,S)		**O/P**			

Step 4(Partitioning Phase): Buffer values written to disk partition. Mod3 as 1 loaded to output buffer

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)

(4,T2),(31,T2),(13,T2)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
(11,S),(4,S),(21,S)		**T1**	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(3,S),(31,S),(13,S)		**T2**	(21,T2),(3,T2),(27,T2)		
(5,S),(27,S),(23,S)		**O/P**			

Step 5 (Partitioning Phase): Buffer values written to disk partition. Mod3 as 2 loaded to output buffer

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(11,T2)(5,T2)(23,T2)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
(11,S),(4,S),(21,S)		**T1**	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(3,S),(31,S),(13,S)		**T2**	(21,T2),(3,T2),(27,T2)	(4,T2),(31,T2),(13,T2)	
(5,S),(27,S),(23,S)		**O/P**			

Step 6 (partitioning Phase): write output buffer values to disk.

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(11,T2)(5,T2)(23,T2)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
(11,S),(4,S),(21,S)		**T1**	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(3,S),(31,S),(13,S)		**T2**	(21,T2),(3,T2),(27,T2)	(4,T2),(31,T2),(13,T2)	(11,T2)(5,T2)(23,T2)
(5,S),(27,S),(23,S)		**O/P**			

Step 7 (Joining Phase): Now use the first partitions of T1 and T2. Re-Hash the first partition of T1 into Buffer and read the values from T2 one by one and output the matching records by same hash function to buffer output.

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(3,T1,T2)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
		T1	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(3,T1)(6,T1) (9,T1)		**T2**	(21,T2),(3,T2),(27,T2)	(4,T2),(31,T2),(13,T2)	(11,T2)(5,T2)(23,T2)
(21,T2)(3,T2)(27,T2)		**O/P**			

Step 8(Joining Phase): Now use the second partitions of T1 and T2. Re-Hash the second partition of T1 into Buffer and read the values from the corresponding T2 partition one by one and output the matching records by same hash function to buffer output.

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(4,T1,T2)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
		T1	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(1,T1)(4,T1)(7,T1)		**T2**	(21,T2),(3,T2),(27,T2)	(4,T2),(31,T2),(13,T2)	(11,T2)(5,T2)(23,T2)
(4,T2)(31,T2)(13,T2)		**O/P**	**(3,T1,T2)**		

Step 9 (Joining Phase): Now use the third partitions of T1 and T2. Re-Hash the third partition of T1 into Buffer and read the values from the corresponding T2 partition one by one and output the matching records by same hash function to buffer output and finally to disk

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
(5,T1,T2)		**S**	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
		T1	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(2,T1),(5,T1),(8,T1)		**T2**	(21,T2),(3,T2),(27,T2)	(4,T2),(31,T2),(13,T2)	(11,T2)(5,T2)(23,T2)
(11,T2)(5,T2)(23,T2)		**O/P**	**(3,T1,T2) (4,T1,T2)**		

Buffer		**R**	(1,R),(2,R),(3,R)	(4,R),(5,R),(6,R)	(7,R),(8,R),(9,R)
		S	(11,S),(4,S),(21,S)	(3,S),(31,S),(13,S)	(5,S),(27,S),(23,S)
		T1	(3,T1), (6,T1), (9,T1)	(1,T1)(4,T1)(7,T1)	(2,T1),(5,T1),(8,T1)
(2,T1),(5,T1),(8,T1)		**T2**	(21,T2),(3,T2),(27,T2)	(4,T2),(31,T2),(13,T2)	(11,T2)(5,T2)(23,T2)
(11,T2)(5,T2)(23,T2)		**O/P**	**(3,T1,T2) (4,T1,T2) (5,T1,T2)**		

Question 12

Question: Give three reasons why a query processing engine might use a sort-merge join instead of a hash join? What are the key differences sort-merge and hash join?

Answer:

Situations for a query processing engine to use sort-merge join instead of has join:

- When the join condition is an inequality condition (like <, <=, >=). This is because hash join cannot be used for inequality conditions and if the data set is large, nested loop is definitely not an option.
- When sorts are required by other operations such as ORDER BY, the optimizer finds it is cheaper to use a sort merge than a hash join.
- When the rows are already sorted, optimizer avoids the first step of sort join operation and find it cheaper to use sort-merge join instead of hash join.

Differences between sort-merge and hash join:

- A "sort merge" join is performed by sorting the two data sets to be joined according to the join keys and then merging them together. The merge is very cheap, but the sort can be prohibitively expensive especially if the sort spills to disk. The cost of the sort can be lowered if one of the data sets can be accessed in sorted order via an index,

although accessing a high proportion of blocks of a table via an index scan can also be very expensive in comparison to a full table scan.

- A hash join is performed by hashing one data set into memory based on join columns and reading the other one and probing the hash table for matches. The hash join is very low cost when the hash table can be held entirely in memory, with the total cost amounting to very little more than the cost of reading the data sets. The cost rises if the hash table has to be spilled to disk in a one-pass sort, and rises considerably for a multi-pass sort.
- Hash joins can only be used for equi-joins, but merge joins are more flexible.
- In general, if you are joining large amounts of data in an equi-join then a hash join is going to be a better bet.
- Sort merge perform better than hash join when the join condition columns are already sorted or there is no sorting required.

Question 13

Question: Let r and s be relations with no indices, and assume that the relations are not sorted. Assuming infinite memory, what is the lowest-cost way (in terms of I/O operations) to compute r ⋈ s? What is the amount of memory required for this algorithm?

Answer: We can store the entire smaller relation in memory. We then read the larger relation block by block and perform nested-loop join using the larger relation as the outer relation.

The number of I/O operations is equal to the sum of the blocks to read from relations r and s i.e., $(B_r + B_s)$

The amount of memory required is number of pages/blocks from smaller relation plus 2 pages (for reading the outer relation block and writing the result). This is min(B_r, B_s) + 2 pages

Question 14

Question:

Rewrite/transform the following query into an equivalent query that would be significantly more efficient.

```
select
people.playerid, people.nameLast, people.throws,
batting.teamid, batting.yearid, ab, h, rbi from
        (people join batting using(playerid))
where teamid='BOS' and yearID='1960';
```

Answer: Performing the selection as early as possible reduces the size of the relation to be joined and thus produces better performance results. Using this query optimization principal, the above query can re-written by pushing down the where clause to sub-query and then join:

```
Select * From
(select playerid, namelast, throws from people) a
join
(Select playerid, teamid, yearid, ab, h, rbi from batting where teamid='BOS' and yearid='1960') b
using(playerid)
```

Question 15

Question: Suppose that a B+-tree index on (dept_name, building) is available on relation department. (Note: This data comes from the database at https://www.db-book.com/db7/index.html)

What would be the best way to handle the given selection?

Answer: Using the index, we locate the first tuple having dept_name = "Music". We then traverse successive leaf nodes of the index (using pointers) for all entries in the leaves that have dept_name as "Music" and building < "Watson" (they are next to each other), until we stop at an entry in a leaf that doesn't satisfy these two conditions. For each of these entries that are traversed, we retrieve the corresponding tuple in the heap. From the tuples retrieved, the ones not satisfying the condition (budget < 55000) are rejected.

Question 16

Question: Consider the following relational algebra expression

$$\pi_{a,b,c}\,(R \bowtie S)$$

This is a project on the result of a natural join on R and S. Assume that column *a* comes from R, column *b* comes from S and that *c* is the join column. Also assume that both R and S have many large columns. Write an equivalent query that will be more efficient, and explain why.

Answer:

$$\pi_{a,b,c}\left(\sigma_{a,c}(R) \bowtie_{R.c=S.c} \sigma_{b,c}(S)\right)$$

We will use an inner join on the column c, so that the resulting table is joined using the corresponding values of column c. We do not want the join to consider other columns with similar names, which is the case with natural join.
Also we use select statement on both the tables first so that only specific columns get selected first and then join operation is performed. This is more effective in lieu of selecting all the columns and then joining and selecting the specific columns.

Question 17

Answer:

Most DBMSs offer a number of *transaction isolation levels*, which control the degree of locking that occurs when selecting data. For many database applications, the majority of database transactions can be constructed to avoid requiring high isolation levels (e.g. SERIALIZABLE level), thus reducing the locking overhead for the system. Please find below the details on three of the isolation levels:

Read Uncommitted:
Read Uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction, thereby allowing dirty reads. In this level, transactions are not isolated from each other.

Consider the below schedule as an example of one that satisfies the read uncommitted isolation level but not is not serializable.

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(A)	

- Transaction T2 reads the values of data item A written by transaction T1 though the transaction T1 is not committed yet.
- The schedule is not serializable as there exists a cycle in the precedence graph. T2 read data written by T1. And T1 reads data written by T2.

Read Committed:

In this isolation level, a lock-based concurrency control DBMS implementation keeps write locks (acquired on selected data) until the end of the transaction, but read locks are released as soon as the SELECT operation is performed (so the non-repeatable Phantom phenomenon can occur in this isolation level). *Range-locks* are not managed.

Putting it in simpler words, read committed is an isolation level that guarantees that any data read is committed at the moment it is read. It simply restricts the reader from seeing any intermediate, uncommitted, 'dirty' read. It makes no promise whatsoever that if the transaction re-issues the read, it will find the same data; data is free to change after it is read.

Consider the below schedule as an example of one that satisfies the read uncommitted isolation level but not is not serializable.

T1	T2
Lock-S(B) Read(X)	
Unlock(B)	Lock-X(B) Write(B) Unlock(B) Commit
Lock-S(B) Read(B) Unlock(B) Commit	

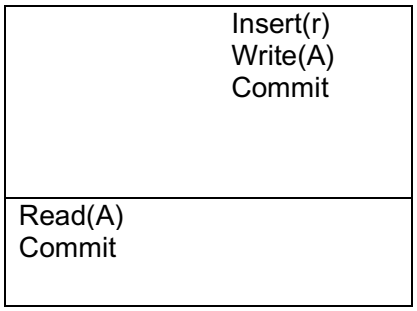
- The first time Transaction T1 reads data item B, it was written before T2. T1 precedes T2 here.
- When T1 reads the data item the second time, it sees the value written by T2. It results in T2 preceding T1.
- There is a cycle in the precedence graph and the schedule is not serializable.

Repeatable Read:

In this isolation level, a lock-based concurrency control DBMS implementation keeps read and write locks (acquired on selected data) until the end of the transaction. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read. Range-*locks* are not managed, so Phantom reads can occur.

Consider the schedule where operation pred_read(t, P) implying that transaction reads all tuples in relation t that satisfy predicate P.

T1	T2
Pred_read(t,P)	



- Let the record 'r' inserted by T2 satisfy the predicate P.
- Since T1 does not see r, insert by T2 implies that T2 precedes T1.
- The read transaction T1 forces T1 precedes T2.
- There is this cycle in precedence graph and so the schedule is not serializable.

Question 18

Question: Explain the difference between a *serial schedule* and a *serializable schedule.*

Answer:

A serial schedule is a schedule where the actions of the same transactions are grouped together. In **Serial schedule**, a transaction is executed completely before starting the execution of another transaction. In other words, you can say that in serial schedule, a transaction does not start execution until the currently running transaction finished execution. This type of execution of transaction is also known as **non-interleaved** execution. The example below is a serial schedule.

```
READ1(A, t)          // Subscript indicate the transaction
   t = t + 100
   WRITE1(A, t)
   READ1(B, t)
   t = t + 100
   WRITE1(B, t)

 READ2(A, s)
   s = 2×s
   WRITE2(A, s)
   READ2(B, s)
   s = 2×s
   WRITE2(B, t)
```

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. A **serializable schedule** is the one that always leaves the database in consistent state.

A schedule S is **serializable schedule** if there is a serial schedule S' such that the effect of execution of S and S' are identical for every DB State. Please find below an example of serializable schedule.

Serializable schedule :

```
   T1                T2                   A        B
=============================================================
   READ(A, t)                             c        c
   t = t + 100
   WRITE(A, t) ------------------->  c+100
                    READ(A, s)
                    s = 2×s
                    WRITE(A, s) ----> 2c+200
   READ(B, t)
   t = t + 100
   WRITE(B, t) ------------------------------> c+100
                    READ(B, s)
                    s = 2×s
                    WRITE(B, s) -------------> 2c+200
```

Is equivalent to the following serial schedule.

```
   T1                T2                   A        B
=============================================================
   READ(A, t)                             c        c
   t = t + 100
   WRITE(A, t) ------------------->  c+100

   READ(B, t)
   t = t + 100
   WRITE(B, t) ------------------------------> c+100
                    READ(A, s)
                    s = 2×s
                    WRITE(A, s) ----> 2c+200
                    READ(B, s)
                    s = 2×s
                    WRITE(B, s) -------------> 2c+200
```

A serial schedule runs the transactions one after another while a serializable schedule interleaves the execution of the transactions.

Question 19

Question: What are the benefits and disadvantages of strict two phase locking?

Answer: In the strict 2 phase locking protocol, locks are obtained and released in two separate phases just like two phase locking protocol. Additionally, it also requires that all exclusive locks should be held by the transaction until it commits. It may release all shared locks after the lock point has been reached.

Advantages:

- Recoverability is easy. Because it produces only cascade-less schedules, it has low rollback overhead and recovery is very easy
- Does not incur cascading aborts.
- Solves dirty read problem. It ensures that any data written by an uncommitted transaction is locked in exclusive mode until the transaction commits.

Disadvantages:

- Limits concurrency. As the set of schedules obtainable is a subset of those obtainable from plain two-phase locking, concurrency is reduced.
- Deadlocks are still possible.

Question 20

Question: Outline the no-steal and force buffer management policies.

Answer:

No-Steal Policy ensures that blocks modified by a transaction that is still active are not written to disk. This is useful for ensuring atomicity without UNDO logging. The no-steal policy does not work with transactions that perform a large number of updates, since the buffer may get filled with updated pages that cannot be evicted to disk, and the transaction cannot then proceed.

Force Policy would ensure transactions would force-output all modified blocks to disk when they commit. It provides durability without REDO logging. It slows down the transaction commit because all the modified blocks are forced to flush to the disk prior to transaction commit. Causes poor performance because of this.

No-Steal/Force policy is the easiest to recover while Steal/No-Force policy has the better performance.