# CSCI 5333 DBMS Classroom Notes

**11/30/2021**

**Introduction to Physical DB**

by K. Yue

**1. Introduction**

* Data in database is stored in storage medium to provide persistence.
	+ What are the storage media?
	+ How are the data stored?
* I/O operations are usually the most significant factor for database operation latency, not CPU or memory operations.

I/O operation / CPU operation = 5 \* 10-3 sec / 2.5 \* 10-10 sec = 107

4GHz => 1/ (4 \* 109) => 2.5 \* 10-10 sec

Access time (read-write arm): 5 ms = 5 \* 10-3 sec

* The [memory hierarchy](http://en.wikipedia.org/wiki/Memory_hierarchy) distinguishes each level of computer storage by access time. Higher level memory is usually:
	+ faster in access time,
	+ lower in volume (size),
	+ more expensive in cost, and
	+ likely to be not persistent: volatile memory.
* Example: [nearline service](https://cloud.google.com/storage-nearline/) by Google.
* Relational models: tuples and relations -> File systems: records and files.
* Principle of locality:
	+ More frequently used data stored in memory of higher level.
	+ Caching may be used to enhance performance.
* Considerations of selecting secondary storage devices:
	+ Speed
	+ Volume
	+ Cost
	+ Reliability
	+ Availability
* Each type of storage devices has important characteristics that should be considered carefully for uses in database architecture.

***Example***:

An article on [flash memory and database](http://www.hansolav.net/blog/content/binary/HowFlashMemory.pdf) lists important characteristics of flash memory for database consideration:

* No in-place update: to rewrite a block, one or more sectors may need to be erased and rewritten. Erasion may take time in the mini-second ranges.
* No mechanical latency: basic operation in micro-seconds, not mini-seconds.
* Limited number of writes before wearing down.
* Asymmetric read/write: write operations may be twice as long as read operations.

**Physical Structures:**

* A tuple can be a virtual record in a file system.
* To improve performance, records can be grouped and stored in *blocks* to maximize the use of the seek operation in a hard disk.
* Blocking factor: the number of records in a block.
* Files can be sequential files or direct access files.
* Examples of important DB file structures:
	+ B+-tree: the *primary key* is used to navigate through an tree index structure to reach linked terminal nodes that store records.
		- Fast sequential access through the primary key: O(n/B) read operation, where n is the number of records and B is the blocking factor.
		- Fast random access through the primary key: O(lg (n/B)).
	+ Hashing: an address is computed from the primary key for storage:
		- Fast random access through the primary key: O(1).
		- Slow sequential access through the primary key.
	+ R-Tree: for spatial data.
* DBMS may allow you to select the physical structures, sometimes known as the database engines.

**2. Indexing**

* An *index* is an access path created to search for records (tuples) more efficiently.
* There is a cost in creating and maintaining an index.
* Cost effectiveness analysis, including profiling, may be used for consideration of index creation.

**MySQL Indexes**

* In MySQL, indices can be created when a table is created: [syntax](http://dev.mysql.com/doc/refman/5.6/en/create-table.html).

***Example:***

The optional index clause:

{INDEX|KEY} [index\_name] [index\_type] (index\_col\_name,...) [index\_option] ...

or

{FULLTEXT|SPATIAL} [INDEX|KEY] [index\_name] (index\_col\_name,...) [index\_option] ...

and

index\_col\_name: col\_name [(length)] [ASC | DESC]

index\_type: USING {BTREE | HASH}

index\_option: KEY\_BLOCK\_SIZE [=] value | index\_type | WITH PARSER parser\_name | COMMENT 'string'

***Example:***

Full text search in MySQL: [functions](http://dev.mysql.com/doc/refman/5.6/en/fulltext-search.html).

In Sakila, we have two tables: film and film\_text, in which film\_text has full indexing:

CREATE TABLE film\_text (
  film\_id SMALLINT NOT NULL,
  title VARCHAR(255) NOT NULL,
  description TEXT,
  PRIMARY KEY  (film\_id),
  FULLTEXT KEY idx\_title\_description (title,description)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;

To search for a film that contains the keyword "teacher" without using full text search.

select film\_id, title, description
from film
where description like '%teacher%';

Possible problems:

* Slow
* Relevance

Using full text indexing and searching:

select MATCH(title,description) AGAINST('teacher') AS score, film\_id, title, description
from film
where MATCH(title,description) AGAINST('teacher')
order BY score DESC;

results in:

ERROR 1214 (HY000): The used table type doesn't support FULLTEXT indexes

The correct way:

select MATCH(title,description) AGAINST('teacher') AS score, film\_id, title, description
from film\_text
where MATCH(title,description) AGAINST('teacher')
order BY score DESC;

Can you guess the algorithm of MySQL used to calculate scores from the results of the query above?

* There is also a separate create index statement: [syntax](http://dev.mysql.com/doc/refman/5.6/en/create-index.html).

Example: Using AdventureWorks, a small data warehouse example created by Microsoft.

-- using AdventureWorks

-- Basic information about AdventureWorks.address.
use AdventureWorks;
desc address;
select count(\*) from address;

-- Q1
select distinct AddressLine1, City, PostalCode
from address
order by PostalCode
limit 300;

-- Q2
select distinct AddressLine1, City, PostalCode
from address
where PostalCode = 98055;

-- Q3
select distinct AddressLine1, City, PostalCode
from address
where PostalCode = 98055 or PostalCode = 83402;

-- Q4
select distinct AddressLine1, City, PostalCode
from address
where PostalCode between 90000 and 92000
limit 300;

-- create index
create index postal\_index on address(postalCode);
desc address;

-- Q1
select distinct AddressLine1, City, PostalCode
from address
order by PostalCode
limit 300;

-- Q2
select distinct AddressLine1, City, PostalCode
from address
where PostalCode = 98055;

-- Q3
select distinct AddressLine1, City, PostalCode
from address
where PostalCode = 98055 or PostalCode = 83402;

-- Q4
select distinct AddressLine1, City, PostalCode
from address
where PostalCode between 90000 and 92000
limit 300;

-- clean up
drop index postal\_index on address;
desc address;

**Introduction to concurrency control and transaction management**

by K. Yue

**1. Concepts of Concurrency Control**

* Modern databases are multi-user systems: many users access the system concurrently.
* The database may be in a system of:
	+ Single computer single CPU (increasingly rare): interleaved processing.
	+ Single computer multiple CPU (core): parallel processing.
	+ Multiple computers: parallel processing.
	+ Cloud computing: massive parallelism.
* A task may not be executed in one shot. It may be divided into many execution *sequences*.
* There are no guarantee of the relative orders of concurrent tasks in an execution *schedule*.
* Without proper control,
	+ Read-write anomaly and write-write anomaly can occur.
	+ Database may become inconsistent.

***Example:***

Transfer $200 from account 1234 to 2345:

<http://en.wikipedia.org/wiki/SQL#Transaction_controls>

Task t1:
-- Assume: account 1234 = $1,000, account 2345: $500

-- Consistent state: total sum of the two accounts: $1,500
UPDATE Account SET amount=amount-200 WHERE account\_number=1234; -- (1) inconsistent state. total sum of the two accounts: $1,300
UPDATE Account SET amount=amount+200 WHERE account\_number=2345; -- (2)

-- Consistent state: total sum of the two accounts: $1,500

Sequence #1:

(1) Task t1 step (1)
(2) Task t1 step (2)

Sequence #2: read-write anomaly

(1) Task t1 step (1)
(2) Task t2 account reports: account 1234: $800, account 2345: $500 (Problem: access data from an inconsistent state): may want to isolate it from external tasks.
(3) Task t1 step (2)

Sequence #3: crash and recovery: atomicity: all or nothing.

(1) Task t1 step (1)
(2) System crashes; task t1 aborted after step (1) (Problem: system ending up in an inconsistent state)

Sequence #4: write-write anomaly

(1) Task t1 step (1)
(2) Task t2 read account amounts, calculate bonus and update accounts. Bonus will be calculated based on account 1234 = $800, account 2345: $500 (Problem: data update in an inconsistent state)
(3) Task t1 step (2)

* Thus, to avoid accessing an inconsistent state, concurrency control is necessary.
* Concurrency control is mainly done by *transaction management*.
* A transaction is a logical unit of database processing that is *atomic*: either the entire transaction is performed, or none of the transaction action is performed. This is the 'all or nothing' property.
	1. Before transaction: consistent
	2. Inside transaction: possibly inconsistent, but isolated from concurrent access
	3. After transaction: consistent state; action permanent.
* This refers to the famous ACID properties in DBMS: e.g. <http://en.wikipedia.org/wiki/ACID>
* ACID properties (from Elmarsi)
	1. Atomicity: A transaction is an atomic unit of processing. It is either performed in its entirety or not performed at all.
	2. Consistency preservation: A correct execution of the transaction must take the database from one consistent state to another. (physical consistency)
	3. Isolation: A transaction should not make its updates visible to other transactions until it is committed. This property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
	4. Durability or permanency: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.
* Implementing ACID can bring *performance degradation*. Thus, for example, some NoSQL provides only 'eventual consistency'.
* Many DBMS provides ACID by means of locking or multi-versioning.
* Basically, in locking, a transaction may have exclusive access to selected data until the transaction is terminated.
* SQL support of transaction management depends on the vendor. It usually includes:
	1. The execution of a single SQL statement is atomic.
	2. The commands START TRANSACTION (or similar) and SAVE TRANSACTION (or similar) may be available to specify the boundary of transactions.
	3. COMMIT makes all data changes in the transaction to become permanent.
	4. ROLLBACK undo all data changes in the transaction (or since the last COMMIT or ROLLBACK).

***Example: from Wikipedia***

<http://en.wikipedia.org/wiki/SQL#Transaction_controls>

START TRANSACTION;
  UPDATE Account SET amount=amount-200 WHERE account\_number=1234;
  UPDATE Account SET amount=amount+200 WHERE account\_number=2345;

IF ERRORS=0 COMMIT;
IF ERRORS<>0 ROLLBACK;

**Application Programmer' responsibility in terms of ACID:**

1. Atomicity: the transaction is either fully committed, or fully rollback. The programmer needs to define the scope and action of the transaction.
2. Consistency: the execution of transaction should keep data consistent. It is the programmer's responsibility to ensure the logic for consistency. The following transaction can be atomic but inconsistent.

START TRANSACTION;
  UPDATE Account SET amount=amount-200 WHERE account\_number=1234;
  UPDATE Account SET amount=amount+400 WHERE account\_number=2345;

IF ERRORS=0 COMMIT;
IF ERRORS<>0 ROLLBACK;

* Isolation: Concurrent accesses will leave the data base consistent. Consistent states within a transaction is isolated from concurrent access. Usually no concern for the application programmers.
* Durability: Once committed, the transaction is finalized. Usually no concern for the application programmers.

**2. MySQL Transaction Management**

* MySQL Manual on TM statements: <http://dev.mysql.com/doc/refman/5.6/en/sql-syntax-transactions.html>
* Autocommit mode: after the execution of a SQL statement, the result is automatically committed.
* START TRANSACTION disables the autocommit mode.
* MySQL supports COMMIT and ROLLBACK.
* It also supports LOCK TABLES and UNLOCK TABLES.

***Example:*** using Sakila

If a customer rents a film (of inventory id 10, say), Sakila needs to perform the following in an atomic transaction:

1. Insert a new row into the rental table with inventory\_id 10.
2. Insert a new row into the payment table with the rental\_id automatically created in (1).

This can be accomplished by the following code:

drop procedure rent\_film;

DELIMITER $$
#   A customer rents a video
create procedure rent\_film(
   IN customer\_id INT,       # cutomer who rents the video
   IN inventory\_id INT,       # inventory video rented
   IN staff\_id INT,          # staff id
   IN rental\_fees DECIMAL(5,2)) # FEES PAID TO RENT THE VIDEO INITIALLY
LANGUAGE SQL
NOT DETERMINISTIC
READS SQL DATA
SQL SECURITY DEFINER

BEGIN

DECLARE rental\_id INT;

START TRANSACTION;
INSERT INTO rental(rental\_date, inventory\_id, customer\_id, return\_date, staff\_id)
VALUES (CURRENT\_TIMESTAMP(), inventory\_id, customer\_id, NULL, staff\_id);

SET rental\_id := LAST\_INSERT\_ID();

INSERT INTO payment(customer\_id, staff\_id, rental\_id, amount, payment\_date)
VALUES(customer\_id, staff\_id, rental\_id, rental\_fees, CURRENT\_TIMESTAMP());

IF (@@error\_count =0) THEN
   COMMIT;
ELSE
   ROLLBACK;
END IF;

END $$

DELIMITER ;

Testing:

CALL rent\_film(1,10,1,1.99);

select \* from rental
order by rental\_id desc
limit 1;

select \* from payment
order by payment\_id desc
limit 1;

Note:

* the use of LAST\_INSERT\_ID() and CURRENT\_TIMESTAMP().
* The stored procedure is basic and does not include error handling code.