School of Economics and Business Administration

Advanced Databases

IUP ISEA

Year 2003-2004

Jérôme Darmont

http://eric.univ-lyon2.fr/~jdarmont/?lang=eng

Outline

- Introduction
- Transaction management
- Performance optimization

Outline

☞**Introduction**

- Transaction management
- Performance optimization

Short track motivation

- The field of databases is not reduced to:
  - Database conceptual design (E/R, UML…)
  - Database interrogation (SQL)
- Lots of other issues exist
  - Database administration, System issues
  - Performance issues
  - Advanced databases (e.g., object-relational databases, XML databases, web-based data warehouses…)

• • •

Detailed outline

- Transaction management
  - Transaction concepts
  - Recovery techniques
  - Concurrency control techniques
- Performance optimization
  - Indexing
  - Materialized views
  - Buffering
  - Clustering
  - Query optimization

Outline

- Introduction

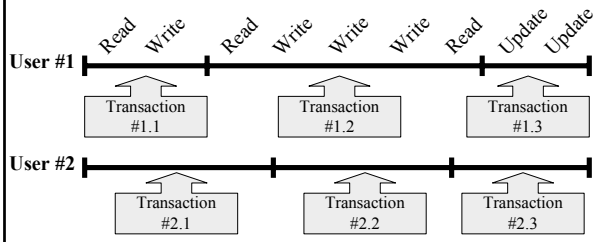
☞**Transaction management**

- Performance optimization

Concept of transaction

- **Transaction:** Logical unit of database processing that includes one or more access operations (read, write, or update)
- A transaction may be
  - **Stand-alone** (e.g., submitted interactively in SQL)
  - **Embedded** with a program (e.g., a PHP script). A program may contain several transactions.

Example of transactions



Note: Transactions may be **concurrent** (i.e., several transactions are executed at the same time).

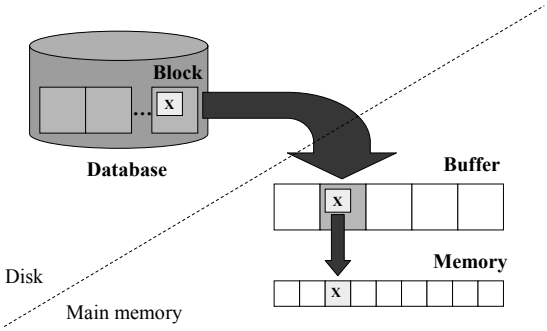
ACID properties of transactions

- **Atomicity:** A transaction is an atomic unit of processing. It is either performed in its entirety, or not performed at all.
- **Consistency:** A correct transaction execution must leave the database in a consistent state.
- **Isolation:** A transaction must not make its updates visible to other transactions until it is entirely and successfully finished.
- **Durability:** Once a transaction changes the database successfully, these changes must never be lost because of subsequent failure.

Transaction operations (1/3)

- **begin_transaction:** Marks the beginning of transaction execution
- **read_item(X):** Reads a database item named X into a program variable (named X too)
  - Finds the address of the disk block (basic I/O unit) that contains item X
  - Copies that block into a buffer in main memory (if it is not already in the buffer)
  - Copies item X from the buffer to variable X

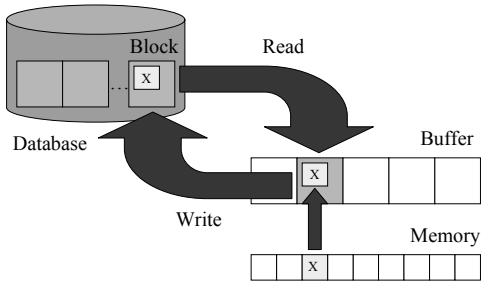
Read operation detail



Transaction operations (2/3)

- **write_item(X):** Writes the value of program variable X into database item X.
  - Finds the address of the disk block that contains item X
  - Copies that block into a buffer in main memory (if it is not already in the buffer)
  - Copies item X from the program variable to its correct location in the buffer
  - Stores the updated block from the buffer back to disk (immediately or later)

Write operation detail



Transaction operations (3/3)

- **end_transaction:** Marks the end limit of transaction execution
- **commit_transaction:** Signals a *successful end* of the transaction. Any update to the database will not be undone.
- **rollback (or abort):** Signals an *unsuccessful end* of the transaction. Any update to the database is canceled.
- **undo:** Cancel one single operation.
- **redo:** Re-execute one single operation.

used for recovery

Need for recovery techniques

What causes a transaction to fail ?

- **Computer failure** (system crash): Everything in memory is lost.
- **Transaction or system error** (logical error): E.g., division by zero, integer overflow…
- **Local error or exception:** E.g., data is not found.
- **Disk failure:** Some disk blocks loose their data when read or written.
- …

System log

- **System log (or journal):** Keeps track of all transaction operations (read, write, commit, etc.) that affect the values of database items
- File that is stored on disk, so that it is not affected by failures except disk failures
- The log is normally periodically backed up to an off-line archive
- Each transaction is associated to a transaction identifier (generated by the system)

System log example

- begin_transaction (T1)
- read_item (T1, X)
- write_item (T1, X, old_value, new_value)
- commit_transaction (T1)
- begin_transaction (T2)
- write_item (T2, X, old_value, new_value)
- write_item (T2, Y, old_value, new_value)
- write_item (T2, Z, old_value, new_value)
- abort (T2)

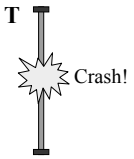
Recovery using log records

Two strategies to recover to a consistent database state in case of system crash:

- **Cancel transaction:** *Undo* every write operation of every unfinished transaction by tracing *backward* through the log and reset all items changed to their *old_value*.
- **Complete transaction:** *Redo* every write operation of every unfinished transaction by tracing *forward* through the log and set all items changed to their *new_value*.

Recovery example

- Transaction T is composed of 100 write operations
- The system log is completely written
- While T is executed, the system crashes after 50 operations are completed



Strategy #1: Cancel
Undo first 50 operations

Strategy #2: Complete
Redo last 50 operations

Concurrency problems (1/2)

- **Lost update problem**
  - Let X be a database item. X = 0.
  - Let T1 and T2 be two transactions that add 1 to X.
  - T1 and T2 are submitted at approximately the same time and their operations are interleaved.
  - read_item (T1, X) // X = 0
  - read_item (T2, X) // X = 0
  - write_item (T1, X, 0, 1) // X = 1
  - write_item (T2, X, 0, 1) // X = 1 instead of 2 !
  - *The updated value resulting from T1 is lost.*

Concurrency problems (2/2)

- **Temporary update (dirty read) problem**
  - Transaction T1 updates a database item
  - Transaction T1 fails
  - The item is accessed by transaction T2 before it is changed back to its original value
  - The item value read by T2 is called *dirty data*
- **Incorrect summary problem**
  - Transaction T1 is calculating an aggregate function (COUNT, SUM, AVG, etc.) on a set of database items
  - Transaction T2 is updating these items at the same time
  - The aggregate function may calculate some values before they are updated and other values after they are updated

Concurrency control techniques

- **Locking** data items to prevent multiple from accessing them concurrently.
- Using **timestamps** (unique transaction identifiers generated by the system)
- …

Locks

- **Lock:** Status variable associated with a data item with respect to possible operations that can be applied to the item.
- **Granularity of locking:** Tuple, set of tuples, table, database.
- **Types of locks:**
  - *Binary locks:* Two states only, too simple, not used in practice
  - *Shared/Exclusive locks:* More general, used in real DBMSs (Database Management Systems)

Binary locks

- **Two states:** *locked* and *unlocked*
- Enforces **mutual exclusion** on the data item: only one transaction can hold the lock at a time and access the item
- Every item in the database is associated to a distinct lock

Operations on binary locks

- **lock_item(X):** Access request to item X
  - If X is already locked, the transaction waits.
  - Otherwise, X is locked and the transaction can proceed.

  A transaction must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed.
- **unlock_item(X):** Release lock on item X
  - X is set to "unlocked".
  - Other transactions can access X.

  A transaction must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed.

Shared/Exclusive locks

- **Three states:**
  - *read-locked (or shared-locked):* Other transactions are allowed to read the item, but not to write it.
  - *write-locked (or exclusive-locked):* No other transaction is allowed to access the item (neither read, nor write).
  - *unlocked*
- **Operations:**
  - *read_lock(X):* Issued before read_item(X)
  - *write_lock(X):* Issued before write_item(X)

Two-phase locking protocol

- **Principle:** All locking operations (read_lock, write_lock) precede the first unlock operation.
- **Phase #1 – Expanding phase:** New locks on items can be acquired, but none can be released.
- **Phase #2 – Shrinking phase:** Existing locks can be released, but no new lock can be acquired
- 2-phase locking enforces transaction **serializability**.

2-phase locking example

- Transaction T1 must read items X and Y
- Transaction T2 must read X and write Y
- T1: read_lock(X) OK
- T1: read_lock(Y) OK
- T2: read_lock(X) OK
- T2: write_lock(Y) Y is locked, T2 waits
- T1: release(X)
- T1: release(Y)
- T2 can proceed in a *new* 2-phase process

Problems with 2-phase locking

- **Holding lock unnecessarily/locking too early:**
  - A transaction T may not be able to release an item X after it has finished with it if T must lock another item Y later on.
  - T must lock Y before it needs it to be able to release X.
- **Penalty to other transactions:**
  - A transaction T' seeking to access X may be forced to wait even though T has finished with X.
  - If Y is locked too early by T, and T' seeks to access Y, T' is forced to wait even though T does not use Y yet.

Deadlocks (1/3)

- **Example of deadlock:**
  - T1: lock(X)
  - T2: lock(Y)
  - T1: lock(Y) T1 waits
  - T2: lock(X) T2 waits too! Problem!
- **Deadlock prevention:**
  - *Conservative locking:* Every transaction locks all the items it needs in advance. If one item cannot be locked, none is locked and the transaction waits. Not used in practice.

Deadlocks (2/3)

- *Item ordering:* A transaction that needs several items locks them according to the order.
  Ex. X is rank 1, Y is rank 2.
  T1: lock(X)
  T2: lock(X) T2 waits
  T1: lock(Y) Deadlock avoided
- *Transaction timestamp:* TS(T) = T starting time
  T1 tries to lock on X but T2 already holds the lock
  Strategy #1 – wait-die: if TS(T1)<TS(T2) then T1 waits
  else abort T1 and restart it later
  Strategy #2 – wound-wait:
  if TS(T1)<TS(T2) then abort T2 and restart it later
  else T1 waits

Deadlocks (3/3)

- *No waiting:* In case of inability to obtain a lock, the transaction aborts and restarts later. Causes many needless aborts.
- *Cautious waiting:*
  if T2 is not blocked then T1 waits
  else abort T1
- *Timeouts:* If a transaction waits for a period longer than a system-defined timeout period, it is assumed to be deadlocked and it is aborted. Problem: some long transactions that are not deadlocked may be aborted.

Starvation (1/2)

- A transaction is **starved** if it waits indefinitely while other transactions are running.
- **Starvation cases:**
  - Waiting scheme for locked item is unfair (some transactions have priority over others)
  - Deadlock prevention algorithm always aborts the same transaction (which thus never ends)
- **Starvation prevention:**
  - *First-come first-serve locking queue:* Fair locking scheme

Starvation (2/2)

- Allow priorities, but increase the priority of a transaction the longer it waits
- Deadlock prevention algorithm affects higher priorities for transactions that have been aborted multiple times
- wait-die and wound-die avoid starvation

Timestamp concurrency control

- Each transaction T is associated to a unique identifier (**timestamp**) TS(T).
- No lock ⇒ no deadlock
- Two timestamps values are associated to each database item X:
  - *Read timestamp:* read_TS(X) = TS(T_Y) where T_Y is the youngest (latest) transaction that has read X successfully;
  - *Write timestamp:* write_TS(X) = TS(T_Y) where T_Y is the youngest (latest) transaction that has written X successfully.

Basic timestamp ordering (1/2)

- **Principle:** Order the transaction based on their timestamp.
- **Write:** T issues a write_item(X) operation

```
if read_TS(X) > TS(T) or write_TS(X) > TS(T) then
    abort(T) and re-run with new timestamp
else if not_reading(X) and not_writing(X) then
    write_item(X)
    write_TS(X) = TS(T)
else
    T waits (waiting queue ordered by timestamp)
end if
```

Basic timestamp ordering (2/2)

- **Read:** T issues a read_item(X) operation

```
if write_TS(X) > TS(T) then
    abort(T) and re-run with new timestamp
else if not_reading(X) then
    read_item(X)
    read_TS(X) = max(TS(T), read_TS(X))
else
    T waits (waiting queue ordered by timestamp)
end if
```


Timestamp CC example (1/2)

- T1 must read items X and Y, TS(T1) = 1
- T2 must read X and write Y, TS(T2) = 2
- T1: read_item(X)
  write_TS(X) = 0 ≤ TS(T1) ⇒ OK
  read_TS(X) = max (TS(T1), read_TS(X)) = 1
- T2: read_item(X)
  write_TS(X) = 0 ≤ TS(T2) ⇒ OK
  read_TS(X) = max (TS(T2), read_TS(X)) = 2

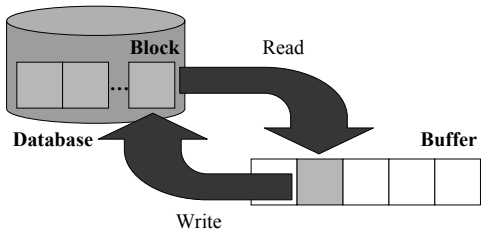
Timestamp CC example (2/2)

- T1: read_item(Y)
  write_TS(Y) = 0 ≤ TS(T1) ⇒ OK
  read_TS(Y) = max (TS(T1), read_TS(Y)) = 1
- T2: write_item(Y)
  read_TS(Y) = 1 ≤ TS(T2) ⇒ OK
  write_TS(Y) = 0 ≤ TS(T2) ⇒ OK
  If T1 has finished reading X:
  write_TS(Y) = TS(T2) = 2
  Otherwise:
  T2 must wait

Outline

- Introduction
- Transaction management
- ☞**Performance optimization**

Input/Output



Block read and write = disk access (I/O)

Performance issue

- Each read/write operation may trigger an I/O
- 1 I/O ≈ several milliseconds
- 1 memory operation ≈ several microseconds

⇒ Minimizing the number of I/Os improves the performances

Sequential access

Example: `SELECT * FROM table WHERE id = id_value`

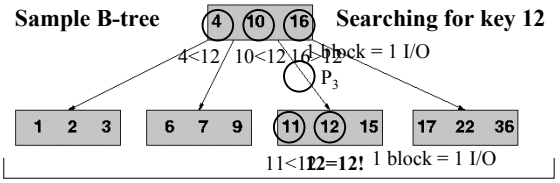
id	Other attributes
1	…
2	…
3	…
4	…
…	
N	…

N/2 read operations on average

($1 + \log_2 N$ with a dichotomy search on *sorted* data)

B-tree indexing

B-tree node: P_1 | K_1 | P_2 | K_2 | P_3 | … | P_{n-1} | K_{n-1} | P_n



Number of I/Os (indices are files) = B-tree *depth* at most
Sequential search in N tuples (M<N blocks) = M/2 I/Os

I/O comparison

- Table of 1.000.000 tuples (N = 1.000.000)
- Record of 1024 bytes (R = 1024)
- Disk block of 4096 bytes (B = 4096)
- Sequential scan: M = N / (B / R)
  = 250.000
  NIO = M / 2
  (on average) = 125.000
- B-tree index: $H \leq \log_T ((N + 1) / 2)$
  T: Min. number of children per node (T = 10)
  (B-tree *degree*) NIO = H ≤ 5,7

Index maintenance and selection

- Indexing all the attributes of a table sounds like a good idea.
- However, each time an new record is inserted in an indexed table, the corresponding index file must be updated. This has a non-negligible cost.
- Indexing all the attributes of a frequently updated table might actually degrades the performances if **maintenance cost** is greater than performance increase.
- **Index selection problem:** Select a set of indices whose maintenance cost is (much) lower than the performance increase they induce.

DBMS Indexing

- B-trees indexing (and extensions, mostly) are intensively used in relational DBMSs
- Extensions include join indices that improve the performance of the costly join operations

Nation index	
UK	1
US	2

Stock index	
UK	10
US	20
US	30

Join index	
1	10
2	20
2	30

- Data warehouses require specific indexing schemes due to their architecture and the huge volumes of data (e.g., *bitmap indices*)

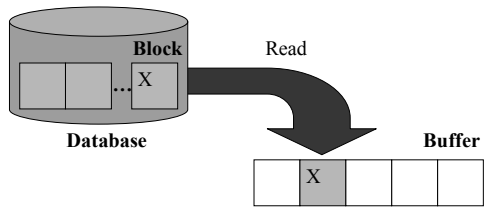
Materialized views (1/2)

- **View** ≈ query result than can be accessed just like a table
- Very useful to pre-compute partial results from a large table, especially aggregates (in data warehouses)
- However, each time a view is accessed, the query is re-executed, which is costly
- **Solution:** *Materialize* the view, i.e., permanently store the result of the associated query (materialized views are tables, in practice)

Materialized views (2/2)

- **Problem:** Materialized view *refreshment* when source data are updated is not automatic
- Refreshment is easy and cheap for non-aggregated data (triggers), but elaborated strategies are needed for aggregated data
- Efficient access to materialized views involves the use of indices
- Materialized views and index selection is a important problem in the data warehousing context

Principle of buffering



Read block X : 1 I/O

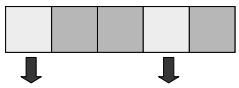
Read block X again : no I/O (X is already in memory)

Block replacement strategies (1/7)

- **Problem:** The buffer is full and a new block must be loaded in memory. What block must be replaced by the new one?

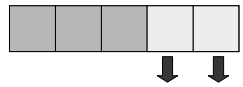
Replacement strategies:

- **RANDOM:** The replaced block is picked up at random. Easy, but not really efficient.



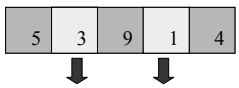
Block replacement strategies (2/7)

- **FIFO** *(First In, First Out)***:** The block that has been in the buffer for the longest time is replaced. Only appropriate to sequential treatments.



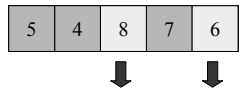
Block replacement strategies (3/7)

- **LFU** *(Least Frequently Used)***:** A usage counter is maintained for each block. The page with the lowest counter value is replaced, whatever its "age".



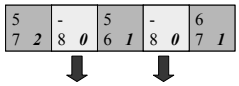
Block replacement strategies (4/7)

- **LRU** *(Least Recently Used)***:** The block that has the oldest usage date is replaced. LRU-queue: when a block is requested, it is placed at the head the queue. It gets out of the queue when time passes, except if it has been used again and placed at the head again. Used a lot.



Block replacement strategies (5/7)

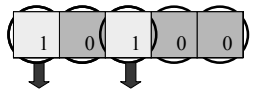
- **LRU-K:** Generalization of LRU with storage of the last K usage dates. Usage distances are cumulated for each block. The block with the highest cumulated usage distance is replaced. The higher K is, the costlier this algorithm is.



K = 2

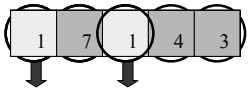
Block replacement strategies (6/7)

- **CLOCK** *("second chance")***:** Variation of FIFO that mimics LRU. Each block is associated to a usage flag that is set to 1 when the page is used. To select the block to replace, all the blocks are scanned in a predefined order (clock concept). The first one to have its flag to 0 is replaced. Blocks with flag set to 1 have it reset to 0, but remain in the buffer.



Block replacement strategies (7/7)

- **GCLOCK** *(Generalized CLOCK)***:** Combines the concepts of LFU and CLOCK. The flag associated to each block is replaced by a usage counter. During "clock scans", these counters are decreased. The first page with a counter set to 0 is replaced. GCLOCK is much better than LFU.

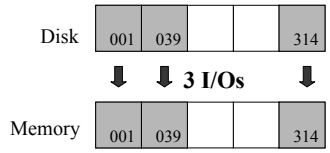


Clustering (1/2)

- **Principle:** Store together on disk data items that are accessed together
- **Data items:** Records or tables
- **Example:**

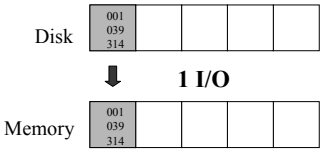
```
SELECT * FROM personne WHERE id IN (001, 039, 314)
```

Without clustering



Clustering (2/2)

With clustering



- Clustering is even more efficient in conjunction with an ad-hoc buffering strategy
- **Problems:**
  - Physical database reorganizations are costly. Clustering *overhead* must be lower than gain.
  - Clustering criterion choice

Query decomposition

- **Problem:** Let's consider the following SQL query over table *employee* with N records

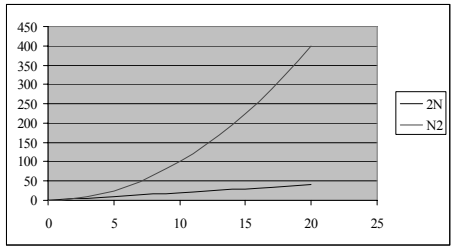
```
SELECT name FROM employee
WHERE salary > (
     SELECT MAX(salary)
     FROM employee)
```

- It can be decomposed into two **sub-queries**
  - `SELECT name FROM employee WHERE salary > constant` (Q1)
  - `SELECT MAX(salary) FROM employee` (Q2)

Cost evaluation (1/2)

- If Q1 is executed first:
  - N read operations to retrieve employees
  - Constant must be recomputed each time (N read operations each time)
  - Total read operations: N^2
- If Q2 is executed first:
  - N read operations to compute max salary
  - N read operations to retrieve employees
  - Total read operations: 2N

Cost evaluation (2/2)



Cost diagram

Order of execution is important!

Query optimization (1/2)

- Query redaction order is important
- **Example:**

```
SELECT * FROM t1 WHERE id_t1 IN ( N1
   SELECT foreign_id FROM T2
   WHERE foreign_id > 100)
```

M2 ≤ N2

provides the same result, but is better than

```
SELECT * FROM t1 WHERE id_t1 IN ( N1
   SELECT foreign_id FROM T2) N2
 WHERE id_t1 > 100
```

N1 * M2 ≤ N1 * N2

Query optimization (2/2)

- Most full DBMSs (DB2, Oracle, SQL Server) include a **query optimizer** that:
  - Decomposes a query into sub-queries
  - Translates the sub-queries into relational algebra
  - Organizes the atomic operations into an execution tree that lists all the possible sequences
  - Uses a *cost model* to find out the best *query execution plan* out of the execution tree
- Others DBMS do not feature a query optimizer (Access, MySQL…). Beware!

Outline

- Introduction
- T**Time for the quiz!**t
- Performance optimization