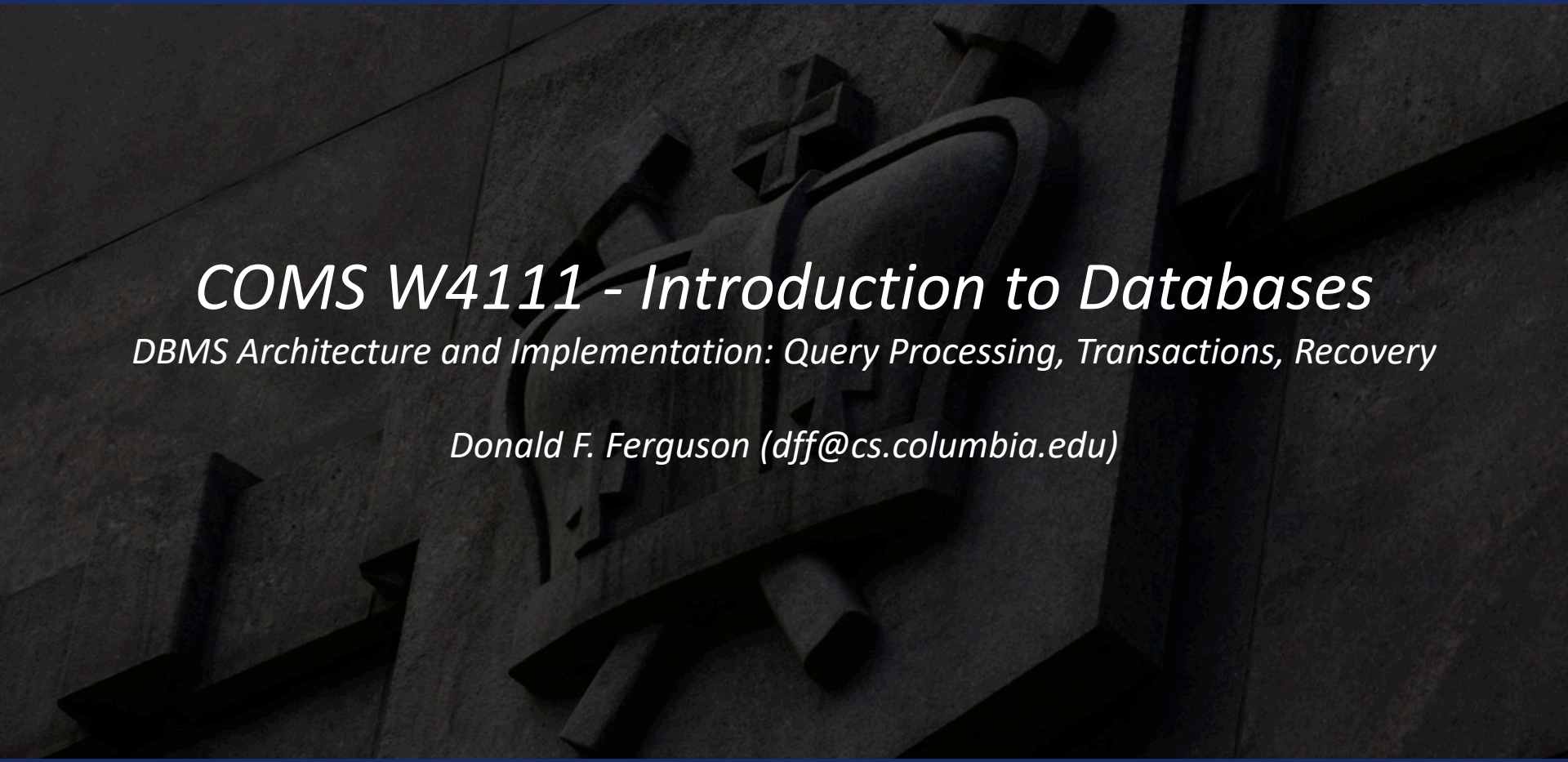


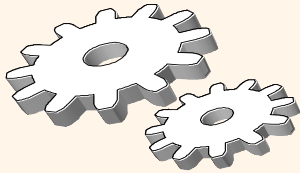
COMS W4111 - Introduction to Databases

DBMS Architecture and Implementation: Query Processing, Transactions, Recovery

Donald F. Ferguson (dff@cs.columbia.edu)

Query Processing Continued

Recap



Overview of Query Evaluation

- *Plan: Tree of R.A. ops, with choice of alg for each op.*
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- Two main issues in query optimization:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: Want to find best plan. Practically: Avoid worst plans!

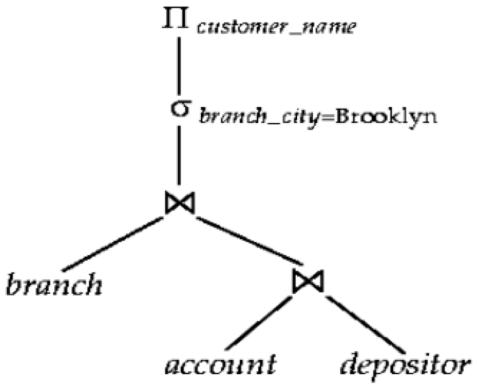
Three Core Optimization Techniques

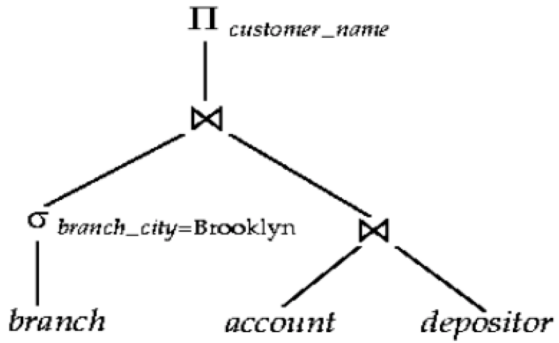
- Query rewrite: Transform the logical query into an equivalent, more efficient query (lower cost query), e.g.
  - R ⋈ S is the same as S ⋈ R
  - $\boldsymbol{\sigma}$(R ⋈ S) is the same at $\boldsymbol{\sigma}$(R) ⋈ $\boldsymbol{\sigma}$(S)
  - $\boldsymbol{\delta}$(R ⋈ S) = $\boldsymbol{\delta}$(R) ⋈ $\boldsymbol{\delta}$(S), where $\boldsymbol{\delta}$ is the distinct operator.
- Access path selection: Which index to choose?
- Operator implementation selection:
  - There are several related implementations for each operator
  - For example,
    - *Sort Scan*
    - *Sort Join*
    - *Hash Join*
    - *Hash Distinct*

Many Ways to Evaluate a Query

Query evaluation

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation





Query Compilation

Database Systems: The Complete Book (2nd Edition) 2nd Edition
by Hector Garcia-Molina (Author), Jeffrey D. Ullman (Author), Jennifer Widom (Author)

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

a) *Parsing.* A *parse tree* for the query is constructed.

b) *Query Rewrite.* The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.

c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Covered in previous lecture.

Part of HW 3.

Will discuss and have an overview in this lecture.

Query Optimization

Parts (b) and (c) are often called the *query optimizer*, and these are the hard parts of query compilation. To select the best query plan we need to decide:

1. Which of the algebraically equivalent forms of a query leads to the most efficient algorithm for answering the query?
2. For each operation of the selected form, what algorithm should we use to implement that operation?
3. How should the operations pass data from one to the other, e.g., in a pipelined fashion, in main-memory buffers, or via the disk?

Each of these choices depends on the metadata about the database. Typical metadata that is available to the query optimizer includes: the size of each relation; statistics such as the approximate number and frequency of different values for an attribute; the existence of certain indexes; and the layout of data on disk.

Will discuss and have an overview in this lecture.

There is A LOT more to query optimization.

Entire semester by itself.

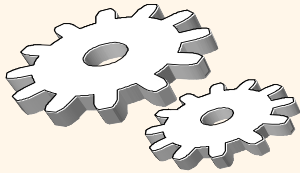
Algorithm Selection

Hash JOIN

- Consider SELECT * FROM people JOIN batting on people.playerID = batting.playerID
- Assume:
  - People has 2 data blocks.
  - Batting has 2 data blocks.
  - I can allocate 3 buffer frames to the JOIN
    - One block holds the current block of R for the scan.
    - One holds a block of S.
    - One holds the JOIN result block I am current constructing.
  - For each row in R, the probability of a buffer hit is 1/2.
  - This means that I will do (1/2) * #(R) I/Os to probe S.
- An alternate approach is to use a hash value on playerID to
  - Read R and partition into 2 buckets, each of size one block.
  - Read S and partition into 2 buckets, each of size one block.
  - I can process the join by loading the 1st block of the hashed R and 1st block of the hashed S into the buffer pool.
  - This means the S tuples matching the current playerID are in the buffer pool.
  - I do some extra upfront I/Os to read, partition and write the hashed tables.
  - But the nested loop is much, much more efficient.
- We still have to scan the "probe" blocks when in memory, but we can hash again.

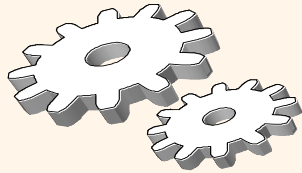
Projection

```
SELECT   DISTINCT
         R.sid, R.bid
FROM     Reserves R
```



- The expensive part is removing duplicates.
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.
- Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.
- If there is an index with both R.sid and R.bid in the search key, may be cheaper to sort data entries!

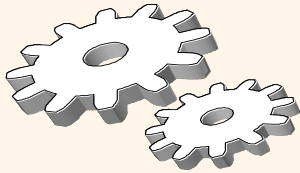
Join: Index Nested Loops



foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)
  - M=#pages of R, p_R=# R tuples per page
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

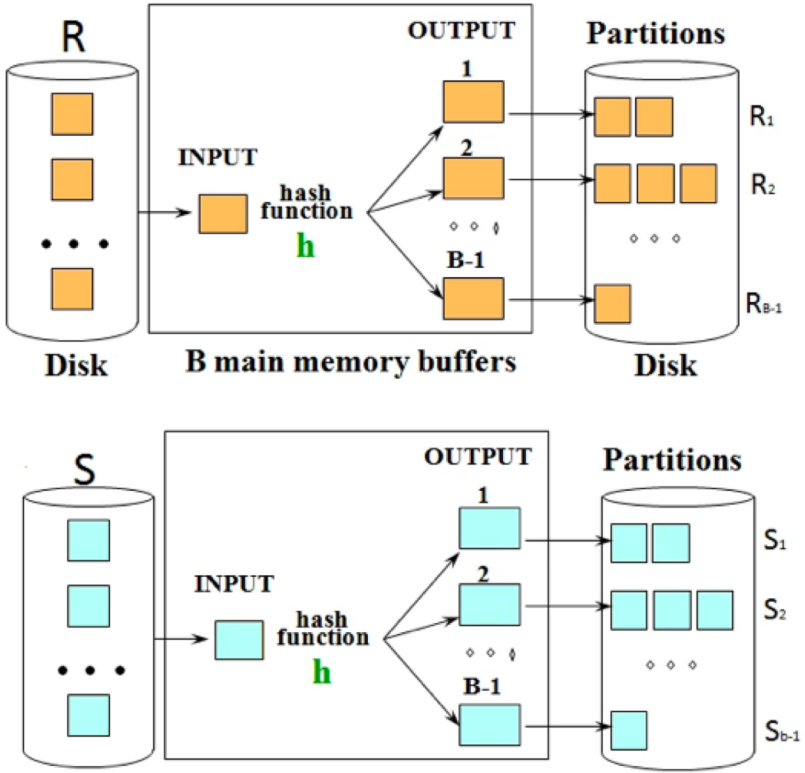
Join: Sort-Merge ($R \bowtie_{i=j} S$)



- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Hash JOIN (http://cs186.wikia.com/wiki/Joins)

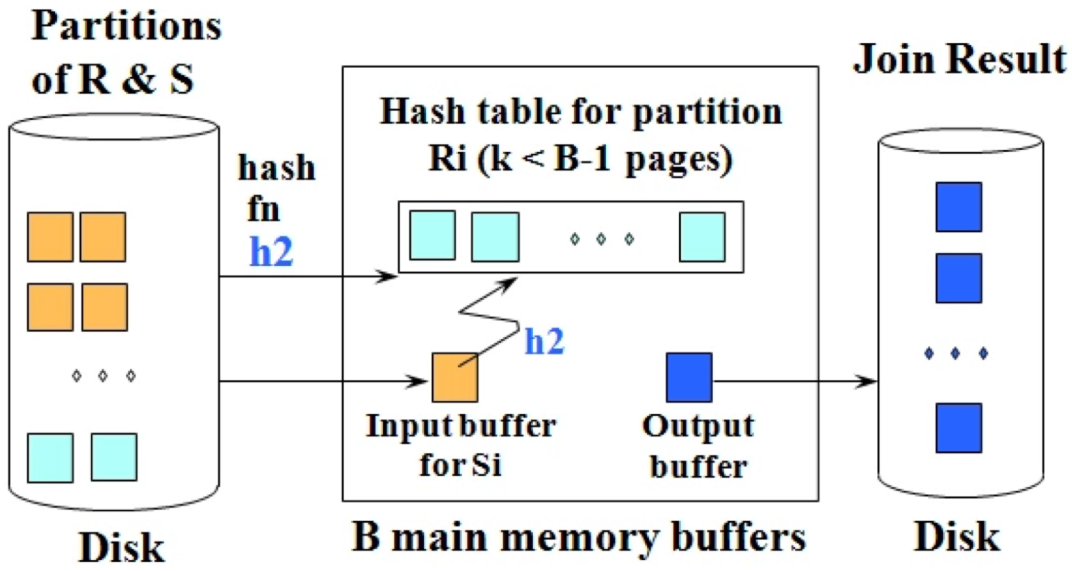
- Hash join uses two stages to accomplish the join.
  - The initial **partitioning** phase,
  - Followed by the **probing** phase.
- In the first stage, we hash the two relations into partitions on disk using a hash function that partitions based on the join condition. The relation *R* gets partitioned into *Ri* and the relation *S* gets partitioned into *Si*. The diagram on the right demonstrates the first stage of the hash join. We will see that in the second stage, we will match *Ri* to *Si* partitions.



Hash JOIN (http://cs186.wikia.com/wiki/Joins)

In the second stage,

- We use an in-memory hash table to perform the joining.
- Read in partition *Ri* and hash it into the in memory hash table
- Scan partition *Si* and probe the in memory hash table for matches.
- Matches go to the output buffer

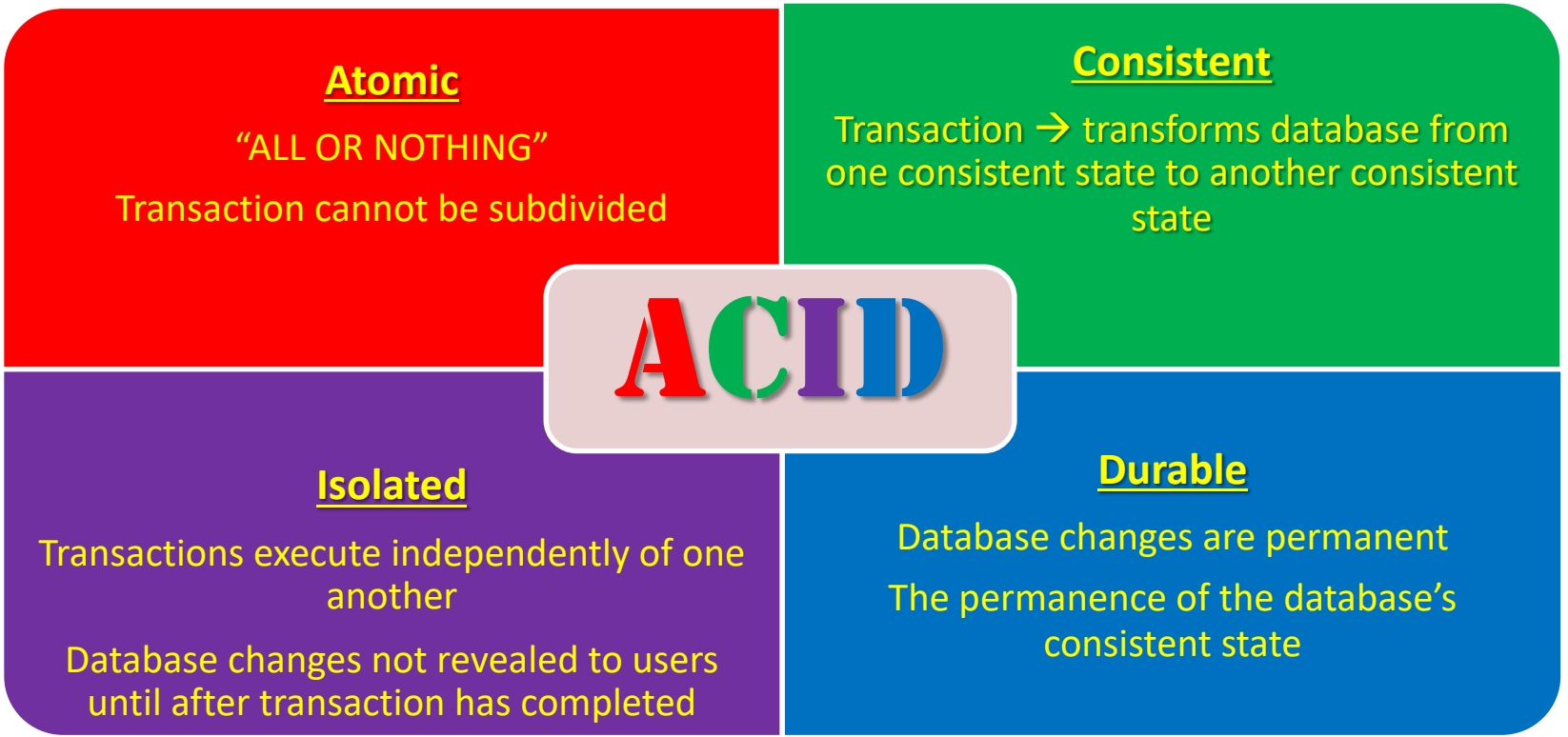


Transactions

Introduction

Core Transaction Concept is ACID Properties

(http://slideplayer.com/slide/9307681)

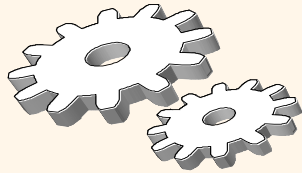


ACID Properties (httpwww.tutorialspoint.com/dbms/dbms_transaction.htm)

A *transaction* is a very small unit of a program and it may contain several low-level tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

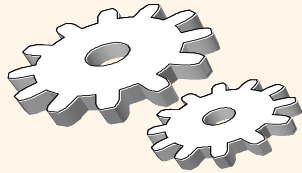
- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

What the Book Says



Transactions

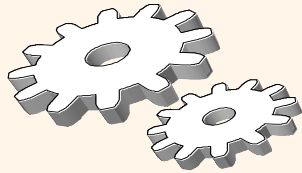
- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.



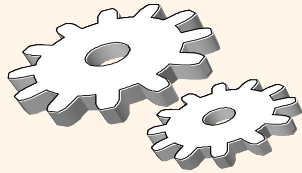
Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Atomicity of Transactions



- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic.* That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

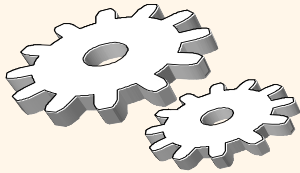


Example

- Consider two transactions (*Xacts*):

```
T1:  BEGIN   A=A+100,   B=B-100   END
T2:  BEGIN   A=1.06*A,   B=1.06*B   END
```

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.



Example (Contd.)

- Consider a possible interleaving (*schedule*):

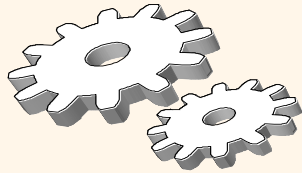
```
T1:   A=A+100,              B=B-100
T2:             A=1.06*A,           B=1.06*B
```

- This is OK. But what about:

```
T1:   A=A+100,                      B=B-100
T2:             A=1.06*A, B=1.06*B
```

- The DBMS's view of the second schedule:

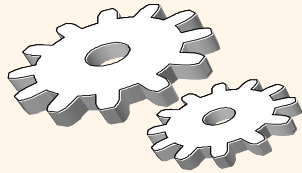
```
T1:   R(A), W(A),                       R(B), W(B)
T2:             R(A), W(A), R(B), W(B)
```



Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules:* For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)



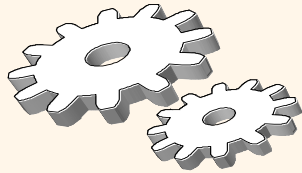
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

```
T1:  R(A), W(A),                    R(B), W(B), Abort
T2:              R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

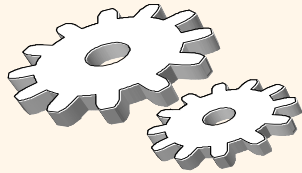
```
T1:  R(A),                  R(A), W(A), C
T2:       R(A), W(A), C
```



Anomalies (Continued)

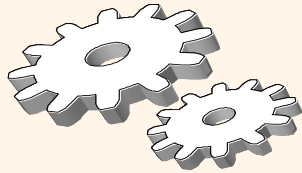
- Overwriting Uncommitted Data (WW Conflicts):

```
T1:  W(A),          W(B), C
T2:       W(A), W(B), C
```



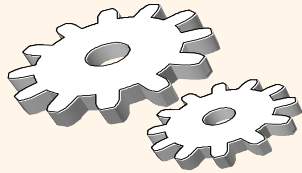
Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing



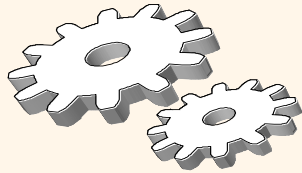
Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.



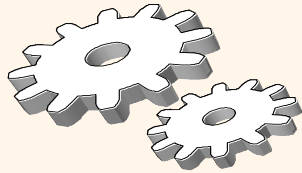
The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.



Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)



Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - *Consistent state*: Only the effects of commited Xacts seen.

Atomicity

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_account.overdraft_limit
3. ELSE
    1. Check that (source_count.balance-amount) >source_account.minimum_balance
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicity

A transaction is a logical unit of work that must be either entirely completed or entirely undone. (All writes happen or none of them happen)

OK. What does this mean? Consider some pseudo-code

def transfer(source_acct_id, target_acct_id, amount)

1. Check that both accounts exist.
2. IF *is_checking_account(source_acct_id)*
    1. Check that (source_acct.balance-amount) > source_acc
3. ELSE
    1. Check that (source_count.balance-amount) >source_ac
4. Update source account
5. Update target account.
6. INSERT a record into transfer tracking table.

Atomicty

- Transaction programs and databases are fast (milliseconds).
  What are the chances of the failure occurring in the wrong spot?
- Well, that doesn't really matter. If it happens,
  - Someone lost money and
  - There is no record off it. Someone is going to very upset.
- Even a small server can have thousands of concurrent transactions ➔
  - There will be corruptions because some transaction will be in the wrong place at the wrong time.
  - Unless we do something in the DBMS
  - Because HW and software inevitably fail
  - And sadly, SW is especially prone to failure when under load

Simplistic Approach

Begin

Update A

Update B

Update C

Commit (REQ)

Force to disk

Commit (RSP)

Simplistic Approach

There are several problems with the simplistic approach.

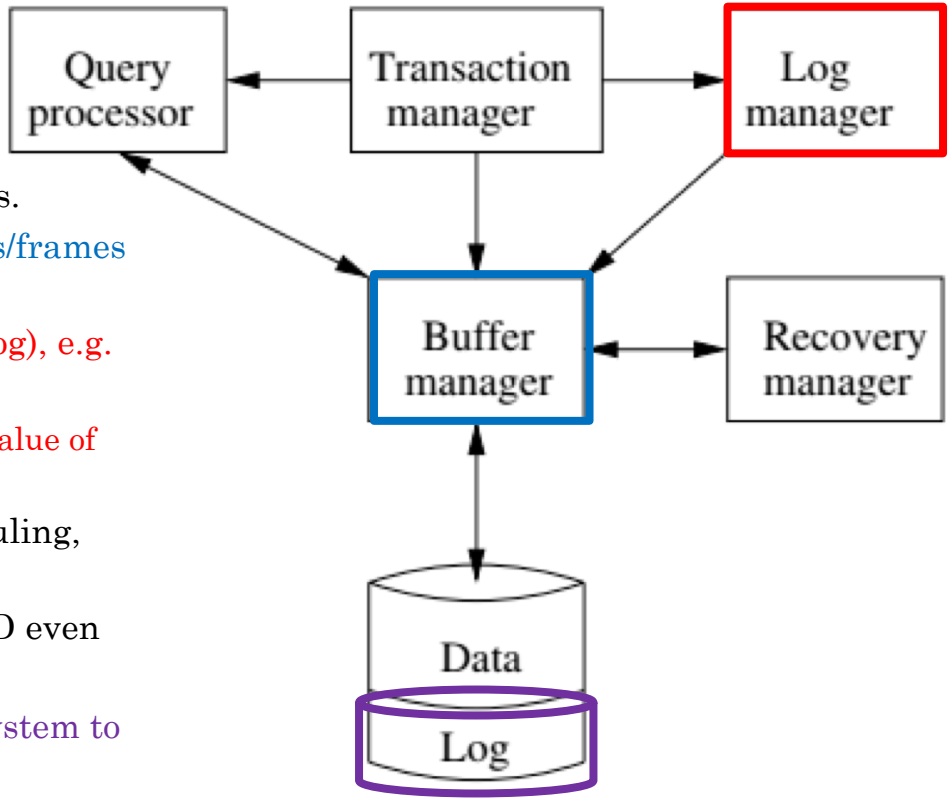
1. The approach does not solve the problem
    1. Some writes might succeed.
    2. Some might be interrupted by the failure, or require retry.
2. Writes may be random and scattered. N updates might
    1. Change a few bytes in N data frames
    2. A few bytes in M index frames

    Transaction rate becomes bottlenecked by write I/O rate, even though a relative small number of bytes change/transaction.
3. Written frames must be held in memory.
    1. Lots of transactions
    2. Randomly writing small pieces of lots of frames.
    3. Consumes lots of memory with pinned pages.
    4. Degrades the performance and optimization of the buffer.
        1. The optimal buffer replacement policy wants to hold frames that will be reused.
        2. Not frames that have been touched and never reused.

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query scheduling, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



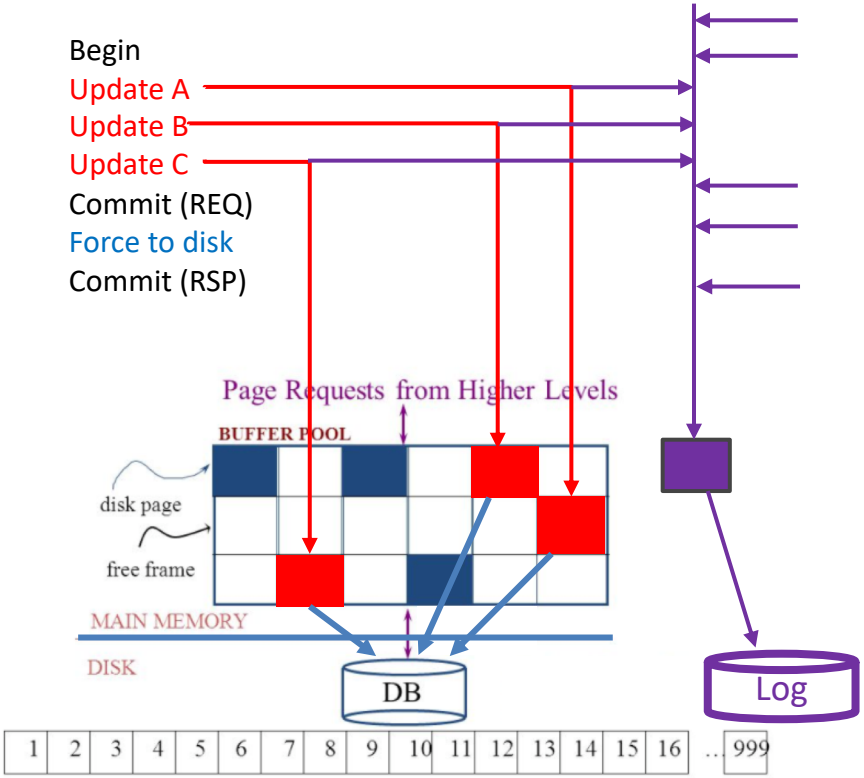
Garcia-Molina et al., p. 846

Logging

The DBMS logs every transaction event

- *Log Sequence Number* (LSN): A unique ID for a log record.
- *Prev LSN*: A link to their last log record.
- *Transaction ID number*.
- *Type*: Describes the type of database log record.
  - **Update Log Record**
    - *PageID*: A reference to the Page ID of the modified page.
    - *Length and Offset*: Length in bytes and offset of the page are usually included.
    - *Before and After Images* of records.
  - **Compensation Log Record**
  - **Commit Record**
  - **Abort Record Checkpoint Record**
  - **Completion Record** notes that all work has been done for this particular transaction.

Write Ahead Logging

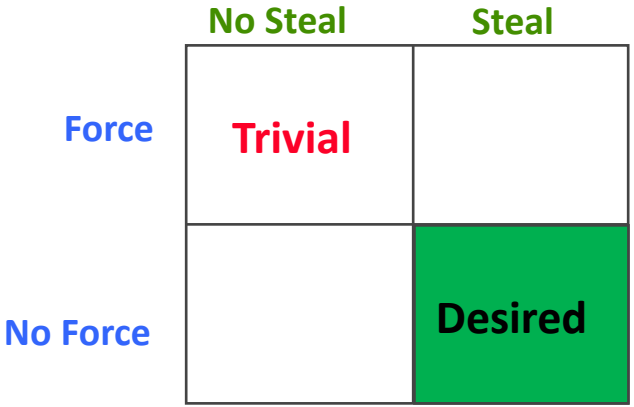


DBMS (Redo processing)

- Write log events from all transactions into a single log stream.
- Multiple events per page
- Forces (writes) log record on COMMIT/ABORT
  - Single block I/O records many updates
  - Versus multiple block I/Os, each recording a single change.
  - All of a transaction's updates recorded in one I/O versus many.
- If there is a failure
  - DBMS sequentially reads log.
  - Applies changes to modified pages that were not saved to disk.
  - Then resumes normal processing.

Write Ahead Logging

- Force every write to disk?
  - Poor response time.
  - But provides durability.
- Steal buffer-pool frames from uncommitted transactions?
  - If not, poor performance/caching performance
  - If yes, how can we ensure atomicity? Uncommitted updates on disk



DBMS (Undo processing)

- Enable steal policy to improve cache performance by
  - Avoiding lots of pinned pages
  - Unlikely to be reused soon.
- Before stealing
  - Force log record to disk.
  - Update log entry has data record
    - Before image
    - After image
- If there is a failure
  - DBMS sequentially reads log.
  - Undoes changes to
    - modified pages, uncommitted pages
    - That were saved to disk.
  - Then resumes normal processing.

ARIES Algorithm =

Algorithms for Recovery and Isolation Exploiting Semantics

ARIES recovery involves three passes

1. Analysis pass:
   - Determine which transactions to undo
   - Determine which pages were dirty (disk version not up to date) at time of
   - RedoLSN: LSN from which redo should start
2. Redo pass:
   - Repeats history, redoing all actions from RedoLSN
     (updated committed but not written changes to pages)
   - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
3. Undo pass:
   - Rolls back all incomplete transactions (with uncommitted pages written to disk).
   - Transactions whose abort was complete earlier are not undone

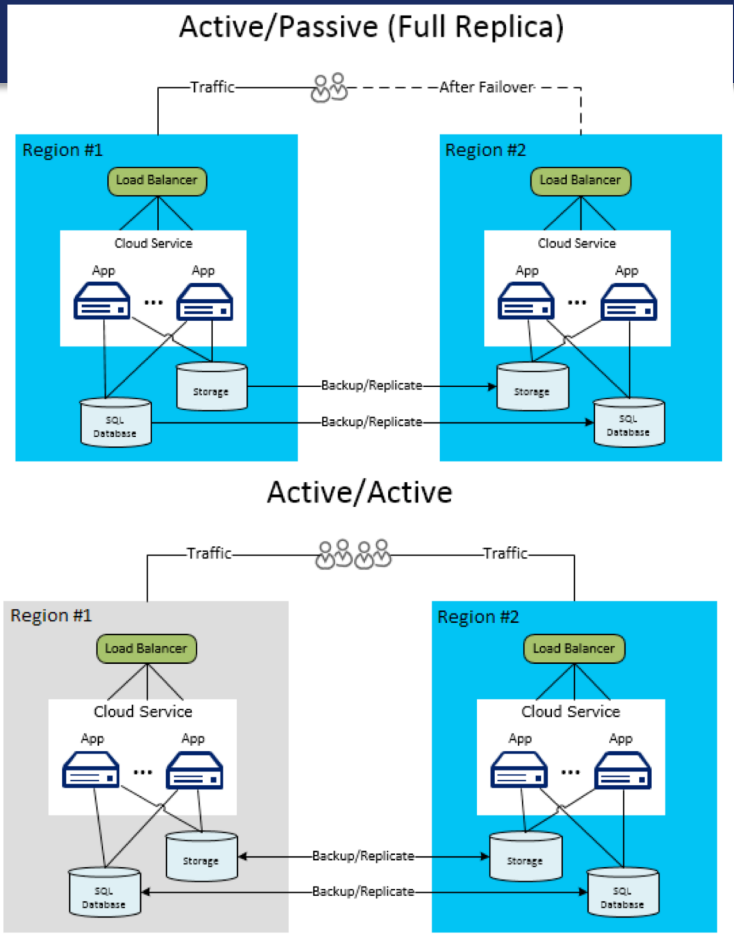
Durability

Durability

- Write changes to disk trivially achieves durability.
- DBMS engine uses write-ahead-logging to
  - Achieve durability
  - But with better performance through more efficient caching and I/O.
- Well, disks fail. How is that durable.
  - RAID and other solutions.
  - Disk subsystems, including entire RAID device, fail →
    - Duplex writes
    - To independent disk subsystems.
- Well, there are earthquakes, floods, etc.

Availability and Replication

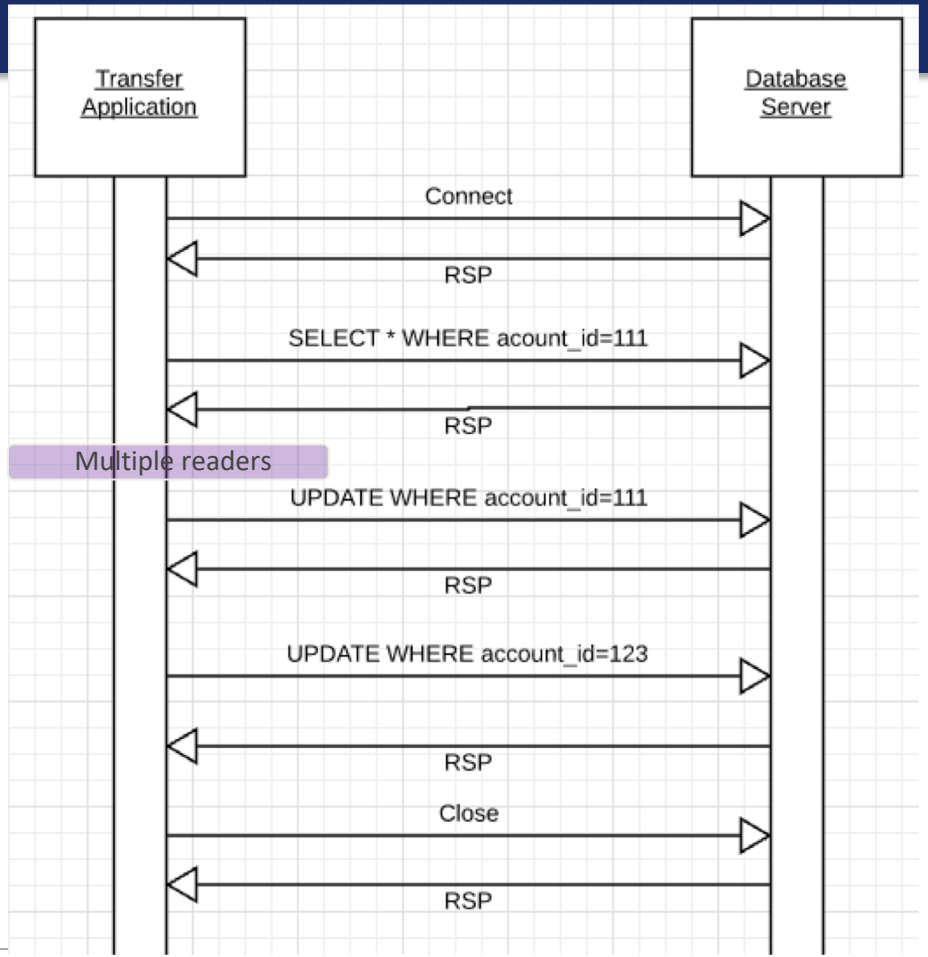
- There are two basic patterns
  - Active/Passive
    - All requests go to *master* during normal processing.
    - Updates are transactionally queued for processing at passive backup.
    - Failure of *master*
      - Routes subsequent requests to *backup*.
      - Backup must process and commit updates before accepting requests.
  - Active/Active
    - Both environments process requests.
    - Some form of distributed transaction commit required to synchronize updates on both copies.
- Multi-system communication to guarantee consistency is the foundation for tradeoffs in CAP.
  - The system can be CAP if and only iff
  - There are never any partitions or system failures
  - Which is unrealistic in cloud/Internet systems.



Isolation

Isolation

- Transfer $50 from
  - account_id=111 to
  - account_id=123
- Requires 3 SQL statements
  - SELECT from 111 to check balance >= $50
  - UPDATE account_id=111
  - UPDATE account_id=123
- There are some interesting scenarios
  - Two different programs read the balance ($51)
  - And decide removing $50 is OK.
- DB constraints can prevent the conflict from happening, but …
  - There are more complex scenarios that constraints do not prevent.
  - Not ALL databases support constraints.
  - The "correct" execution should be that
    - One transaction responds "insufficient funds"
    - Before attempting transfer instead of after attempting.



Isolation

- Consider two **simultaneous** transfer transactions T1 and T2.
- There are two equally ***correct*** executions
  1. T1 transfers, T2 responds "insufficient funds" and does not attempt transfer
  2. T2 transfers, T1 responds "insufficient funds" and does not attempt transfer
- Each correct simultaneous execution is equivalent to a serial (sequential) execution schedule
  - (1) Execute T1, Execute T2
  - (2) Execute T2, Execute T1
- NOTE:
  - We are focusing on ***correctness*** not
  - ***Fairness:***
    - We do not care which transaction was actually submitted first.
    - And probably do not know due to networking, etc.

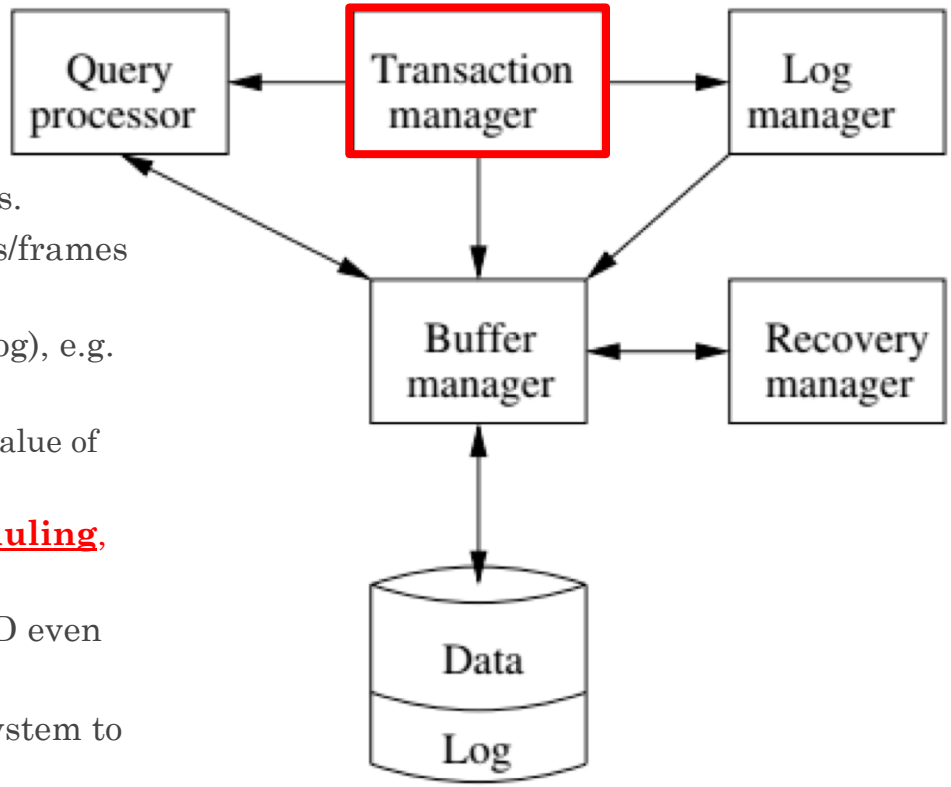
Serializability

"In concurrency control of databases,[1][2] transaction processing (transaction management), and various transactional applications (e.g., transactional memory[3] and software transactional memory), both centralized and distributed, a transaction schedule is **serializable** if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, i.e. without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems."
(https://en.wikipedia.org/wiki/Serializability)

DBMS ACID Implementation

Implementation Subsystems

- *Query processor* schedules and executes queries.
- *Buffer manager* controls reading/writing blocks/frames to/from disk.
- *Log manager* journals/records events to disk (log), e.g.
  - Transaction start/commit/abort
  - Transaction update of a block and previous value of data.
- *Transaction manager* coordinates query **scheduling**, buffer read/write and logging to ensure ACID.
- *Recovery manager* processes log to ensure ACID even after transaction or system failures.
- *Log* is a special type of block file used by the system to optimize performance and ensure ACID.



Garcia-Molina et al., p. 846

Schedule

18.1.1 Schedules

A *schedule* is a sequence of the important actions taken by one or more transactions. When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk. That is, a database element A that is brought to a buffer by some transaction T may be read or written in that buffer not only by T but by other transactions that access A.

T_1	T_2
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

Figure 18.2: Two transactions

Garcia-Molina et al.

Serializable

- Assume there are three
  - concurrently executing transactions
  - T1, T2 and T3
- The transaction manager
  - Enables concurrent execution
  - But schedules individual operations
  - To ensure that the final DB state
  - Is *equivalent* to one of the following schedules
    - T1, T2, T3
    - T1, T3, T2
    - T2, T1, T3
    - T2, T3, T1
    - T3, T1, T2
    - T3, T2, T1

Concurrent execution was *serializable*.

18.1.2 Serial Schedules

A schedule is *serial* if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on. No mixing of the actions is allowed.

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Figure 18.3: Serial schedule in which T_1 precedes T_2

Serializability (en.wikipedia.org/wiki/Serializability)

- **Serializability** is used to keep the data in the data item in a consistent state. Serializability is a property of a transaction schedule (history). It relates to the *isolation* property of a database transaction.
- **Serializability** of a schedule means equivalence (in the outcome, the database state, data values) to a *serial schedule* (i.e., sequential with no transaction overlap in time) with the same transactions. It is the major criterion for the correctness of concurrent transactions' schedule, and thus supported in all general purpose database systems.
- **The rationale behind serializability** is the following:
  - If each transaction is correct by itself, i.e., meets certain integrity conditions,
  - then a schedule that comprises any *serial* execution of these transactions is correct (its transactions still meet their conditions):
    - "Serial" means that transactions do not overlap in time and cannot interfere with each other, i.e, complete *isolation* between each other exists.
    - Any order of the transactions is legitimate, (…)
    - As a result, a schedule that comprises any execution (not necessarily serial) that is equivalent (in its outcome) to any serial execution of these transactions, is correct.

Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

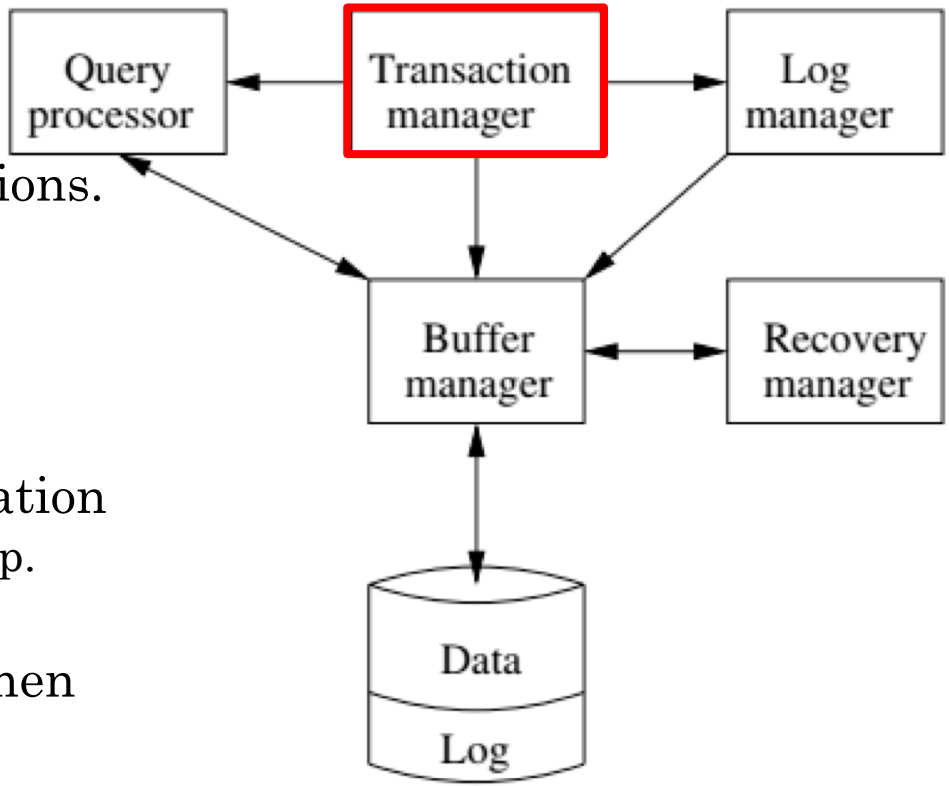
Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

Locking

Transaction Manager

- Intercepts all database operations.
- Acquires/checks locks on
  - Records
  - Pages
  - Index pages
- Suspends and queues an operation
  - In another active, executing op.
  - Has a conflicting lock.
- Restarts queued operations when conflicting locks are released.



García-Molina et al., p. 846

MySQL (Locking) Isolation

13.3.6 SET TRANSACTION Syntax

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic:
    ISOLATION LEVEL level
  | READ WRITE
  | READ ONLY

level:
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
```

Scope of Transaction Characteristics

You can set transaction characteristics globally, for the current session, or for the next transaction:

- With the `GLOBAL` keyword, the statement applies globally for all subsequent sessions. Existing sessions are unaffected.
- With the `SESSION` keyword, the statement applies to all subsequent transactions performed within the current session.
- Without any `SESSION` or `GLOBAL` keyword, the statement applies to the next (not started) transaction performed within the current session. Subsequent transactions revert to using the `SESSION` isolation level.

Isolation Levels (https://en.wikipedia.org/wiki/Isolation_(database_systems)

Not all transaction use cases require 2PL and serializable execution. Databases support a a set of levels.

- **Serializable**
  - With a lock-based concurrency control DBMS implementation, serializability requires read and write locks (acquired on selected data) to be released at the end of the transaction. Also *range-locks* must be acquired when a SELECT query uses a ranged *WHERE* clause, especially to avoid the ***phantom reads*** phenomenon.
  - *The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins.*
- **Repeatable reads**
  - 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. However, *range-locks* are not managed, so ***phantom reads*** can occur.
  - Write skew is possible at this isolation level, a phenomenon where two writes are allowed to the same column(s) in a table by two different writers (who have previously read the columns they are updating), resulting in the column having data that is a mix of the two transactions.[3][4]
- **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 reads phenomenon*** can occur in this isolation level). As in the previous 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.
- **Read uncommitted**
  - This is the *lowest* isolation level. In this level, ***dirty reads*** are allowed, so one transaction may see *not-yet-committed* changes made by other transactions

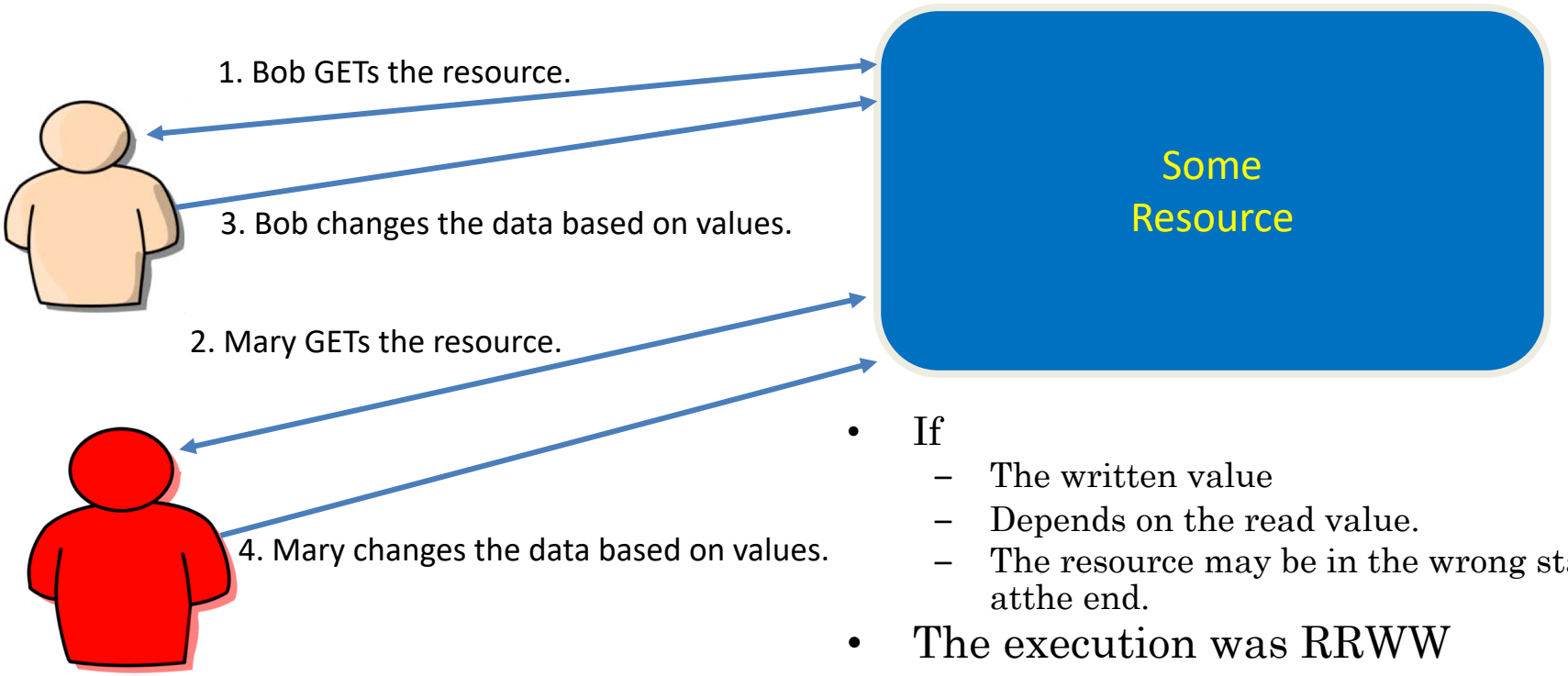
In Databases, Cursors Define *Isolation*

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)
ctx.lookup("jdbc/MyBase");
Connection con = ds.getConnection();
DatabaseMetaData dbmd = con.getMetaData();
if (dbmd.supportsTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)
{ Connection.setTransactionIsolation(TRANSACTION_SERIALIZABLE); }
```

- We have talked about ACID transactions
  - Atomic: A set of writes all occur or none occur.
  - Durable: The data "does not disappear," e.g. write to disk.
  - Consistent: My applications move the database between consistent states, e.g. the transfer function works correctly.
- Isolation
  - Determines what happens when two or more threads are manipulating the data at the same time.
  - And is defined relative to where cursors are and what they have touched.
  - Because the cursor movement determines *what you are reading or have read.*
- *But, … Cursors are client conversation state and cannot be used in REST.*

Read then Update Conflict

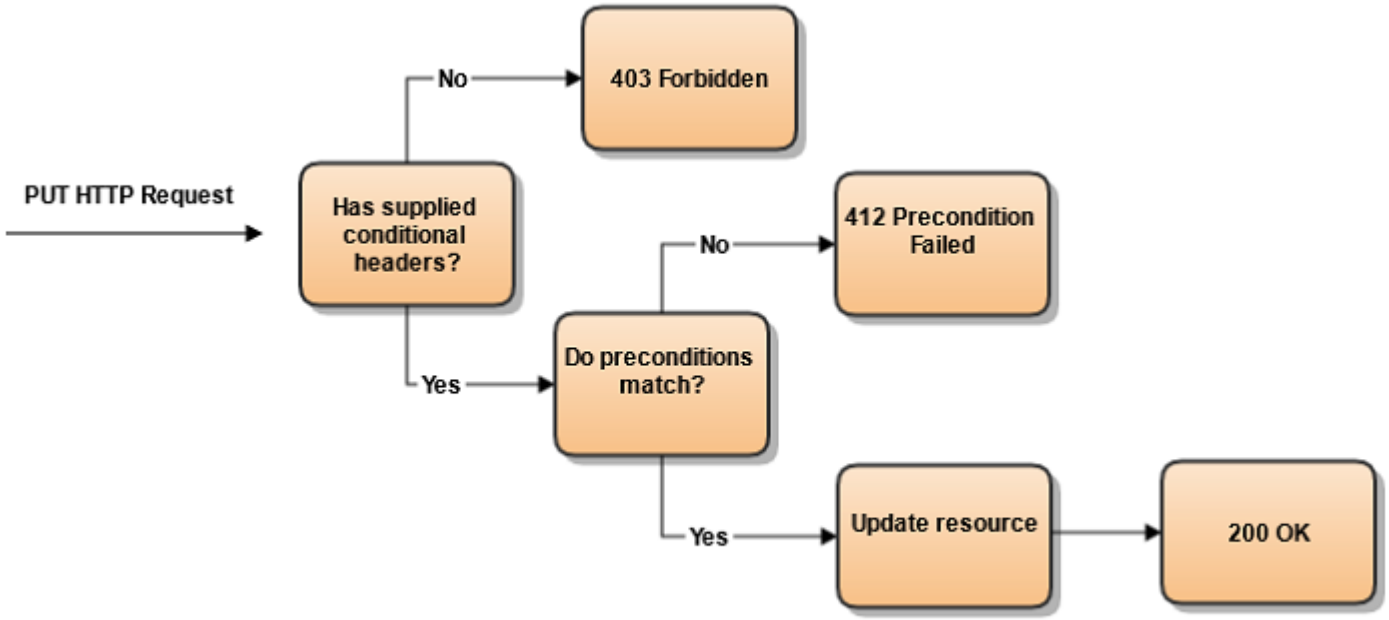


- If
  - The written value
  - Depends on the read value.
  - The resource may be in the wrong state atthe end.
- The execution was RRWW
- But the intent was RWRW.

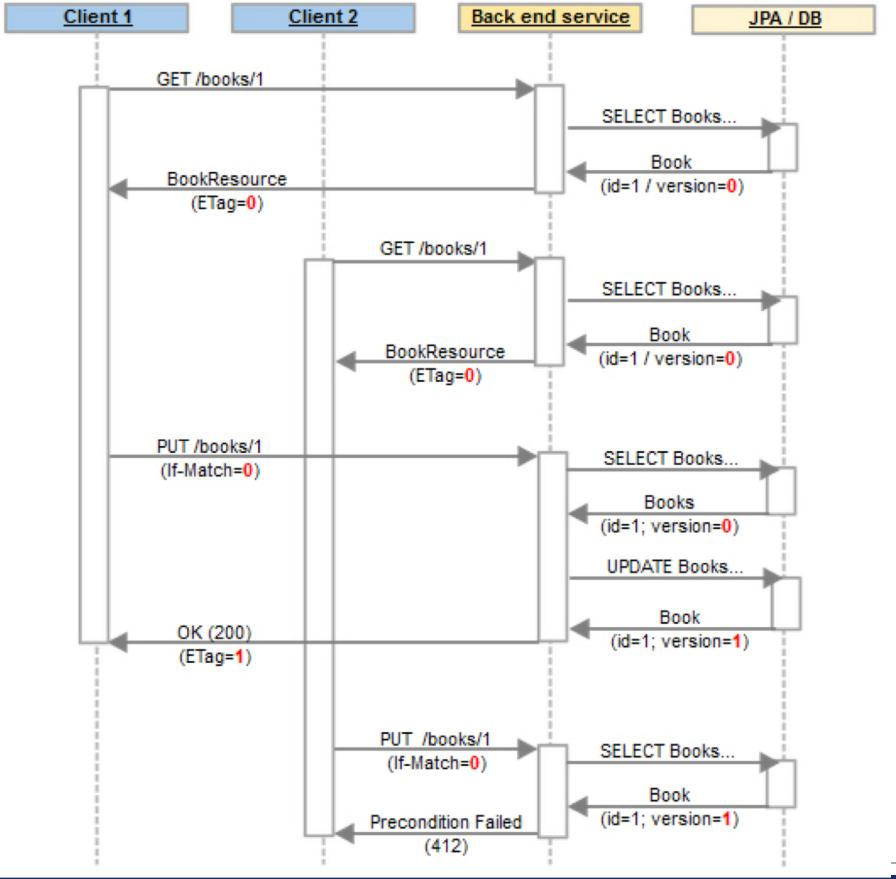
Isolation/Concurrency Control

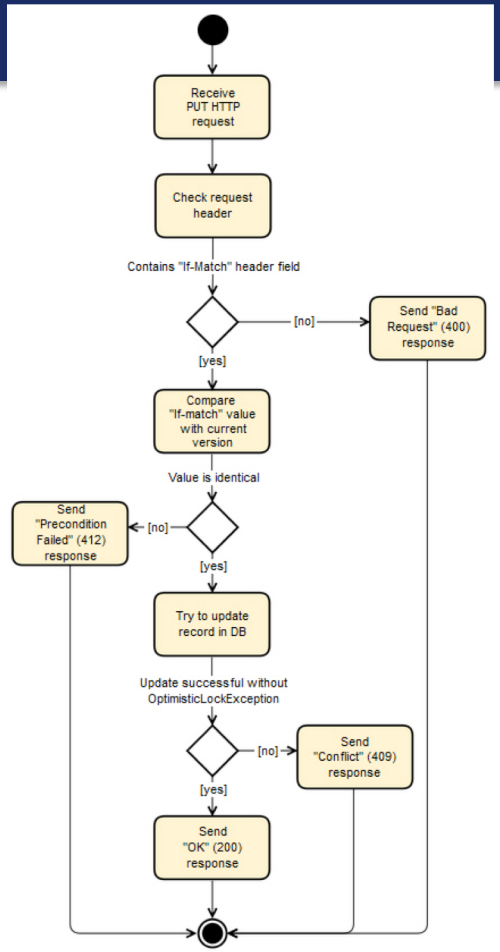
- There are two basic approaches to implementing isolation
  - Locking/Pessimistic, e.g. cursor isolation
  - Optimistic: Before committing, each transaction verifies that no other transaction has modified the data it has read.
- How does this work in REST?
  - The server maintains an ETag (Entity Tag) for each resource.
  - Every time a resource's state changes, the server computes a new ETag.
  - The server includes the ETag in the header when returning data to the client.
  - The server may *optionally* maintain and return "Last Modified" time for resources.
- Semantics on updates
  - If-Match – value of a previous calls ETag response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412.
  - If-Modified-Since – value of a previous Last-Modified response used with a GET. Server should only provide a response if the resource was modified since the timestamp submitted. Use in conjunction with If-None-Match in case the change occurred within the same second. Otherwise provide a 304.
  - If-None-Match – value of a previous calls ETag response used with a GET. Server should only provide a response if the ETag doesn't match, i.e. the resource has been altered. Otherwise provide a 304.
  - If-Unmodified-Since – value of a previous Last-Modified response used with a PUT or DELETE. Server should only act if nobody else has modified the resource since you last fetched it. Otherwise provides a 412

Conditional Processing



ETag Processing





Transaction Example

MySQL Transaction Operations

13.3.1 START TRANSACTION, COMMIT, and ROLLBACK Syntax

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

Switch to Notebook