COSC 460 Lecture 18: Recovery 1

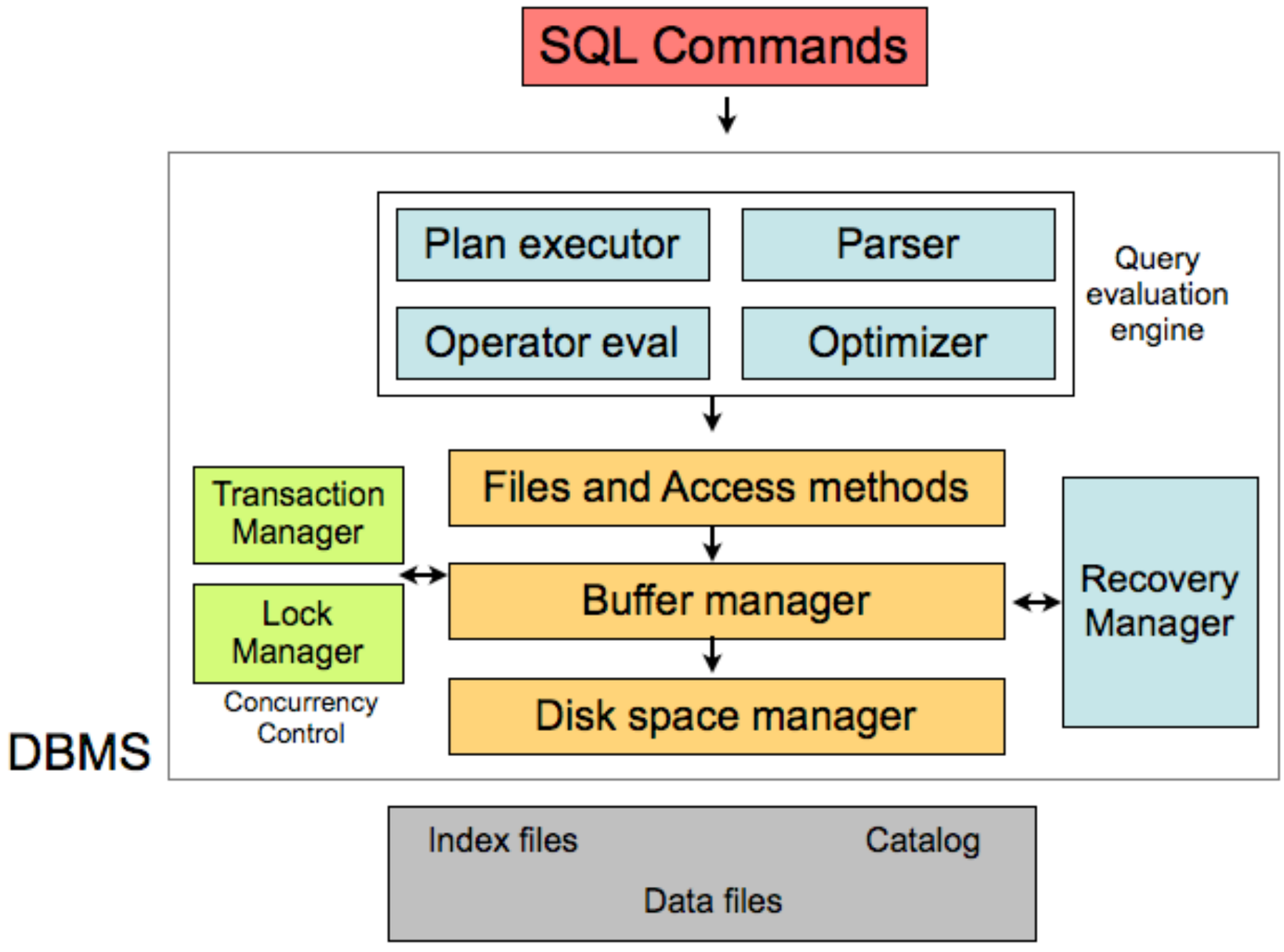
Professor Michael Hay
Fall 2018

Transactions

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Atomicity (and **durability**) have to do with how the DBMS handles *failures*

Architecture of DBMS



(Expected) Failures

- Aborted transaction
- System crash: CPU halts, RAM lost, disk ok
- Other kinds of failures are possible (like what?)… but not directly addressed by DBMS recovery system

Example shown on board

Operators

- IN(x): fetch page containing x from disk into RAM
- OUT(x): flush x's page from RAM to disk
- R(x): do IN(x) if necessary, write value of x to local variable
- W(x): do IN(x) if necessary, write value of local variable x to x's page (*in RAM*)

Key ideas and questions

- Key ideas
  - **Redundancy**: store information twice
  - **Logging**: record *changes* to DB in an *append only* log
- Key questions
  1. What info is written to log?
  2. What is ***logging protocol****?*
  3. What is ***recovery protocol****?*

Approaches

- Approaches

    1. **Undo logging**

    2. **Redo logging**

    3. **Undo/redo logging**

- Why study three?

    - The first two are flawed. The last one works — described in reading and the one you will implement!

    - Studying flaws of first two help explain complexity of last.

Undo Logging

Uncommitted changes made to DB can be rolled back using log.

log

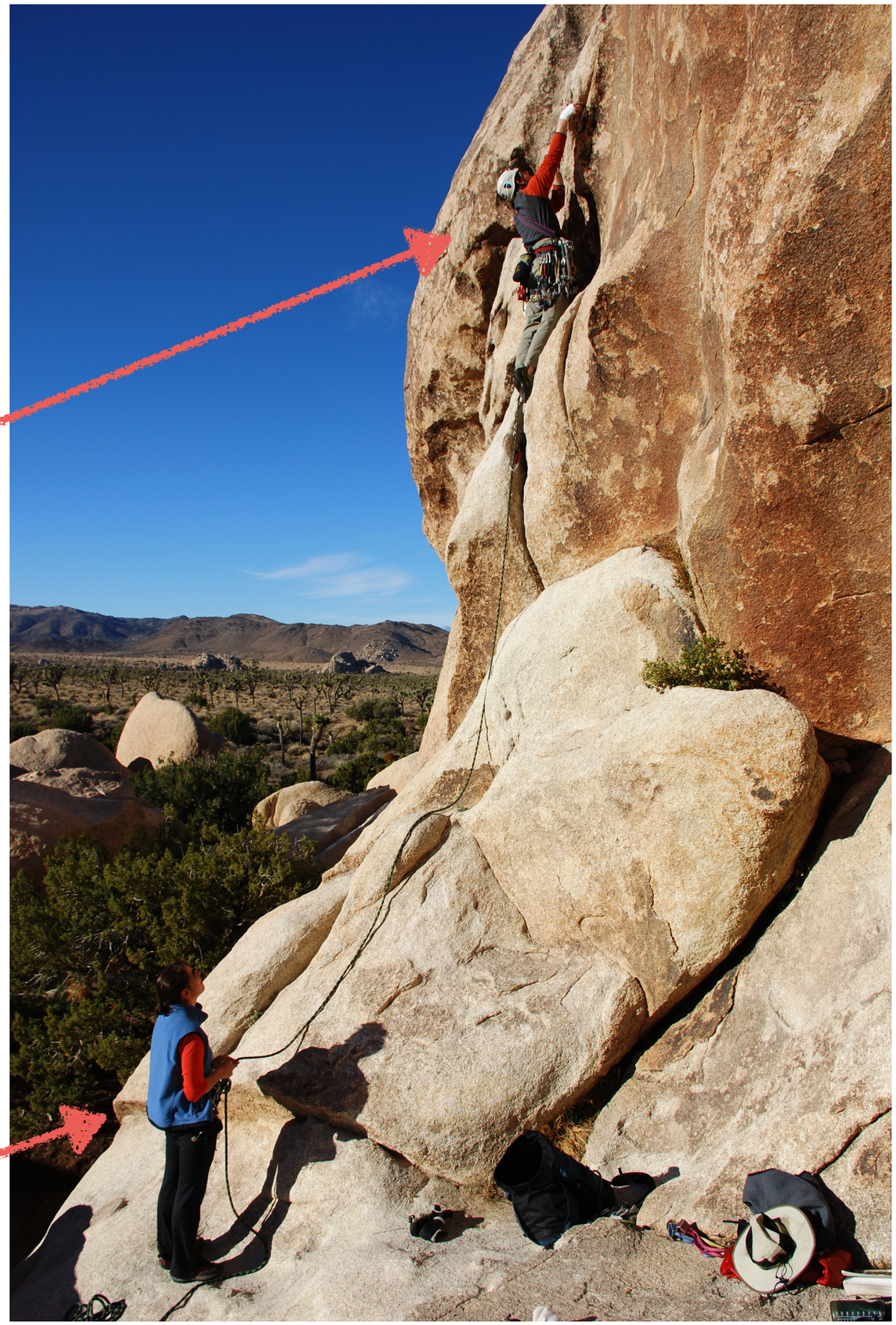
DB

Redo Logging

log

DB

Committed changes recorded in log can be used to update lagging DB.



Logged information

- Types of log records:
  - Transaction start: <T_i, start>
  - Update of data item A:
    <T_i, A, old value → new value >
  - Compensating log record (CLR):
    <CLR, T_i, A, new value >
  - Transaction end successfully: <T_i, commit>
  - Transaction end unsuccessfully: <T_i, abort>

Possible bad states

- Bad State #1: DB changes flushed to disk, but log is still in RAM.

- Bad State #2: Transaction commit flushed to log, but DB changes still in RAM.

- Must devise ***logging protocol*** to avoid bad states.

Undo logging protocol

1. For each DB update, generate log record
2. Write ahead logging: before OUT(X), flush log records up to and including modifications of X.
3. Force: before <T_i, commit> to log, flush all pages dirtied by T_i

Undo recovery protocol

1. Let losers be transactions with start but no commit/abort
2. For each log record from *last* to *first*:
    1. If record was update
       <T_i, A, old → new >
       and T_i is loser, then X = old, W(X), OUT(X)
3. For each T_i in losers, write <T_i, abort> to log

Undo logging

Instructions: ~1 minute to think/ answer on your own; then discuss with neighbors; then I will call on one of you

Suppose a crash occurs and the log and DB are as shown. Use the undo recovery protocol to restore the DB.

(Challenge) The log records actually contain extra information that is never used during recovery. *What is extra?*

Log

```
<T1 start>

<T2 start>

<T3 start>

<T3 B 8→12>

<T1 A 8→16>

<T2 A 16→32>

<T3 commit>
```

DB

A: 16

B: 12

Undo logging

Instructions: ~1 minute to think/ answer on your own; then discuss with neighbors; then I will call on one of you

[Same example as previous question]

Recovery protocol does not specify the order in which abort log records should be written.

Suppose we we write abort in order of transaction id from smallest to largest. (So T1 *before* T2.)

What could go wrong with this example? (Hint: consider the possibility of a crash during recovery.)

Log

```
<T1 start>

<T2 start>

<T3 start>

<T3 B 8→12>

<T1 A 8→16>

<T2 A 16→32>

<T3 commit>
```

DB

A: 16

B: 12

Write out abort messages in order from last to modify to first to modify.

Exercise

Instructions: ~1 minute to think/ answer on your own; then discuss with neighbors; then I will call on one of you

Consider this log and imagine executing the recovery protocol. Something's not right. What? Hint: is this schedule possible under 2PL? Under strict 2PL?

Log

```
<T1 start>

<T2 start>

<T2 A 8→16>

<T1 A 16→32>

<T1 commit>
```

DB

A: 32

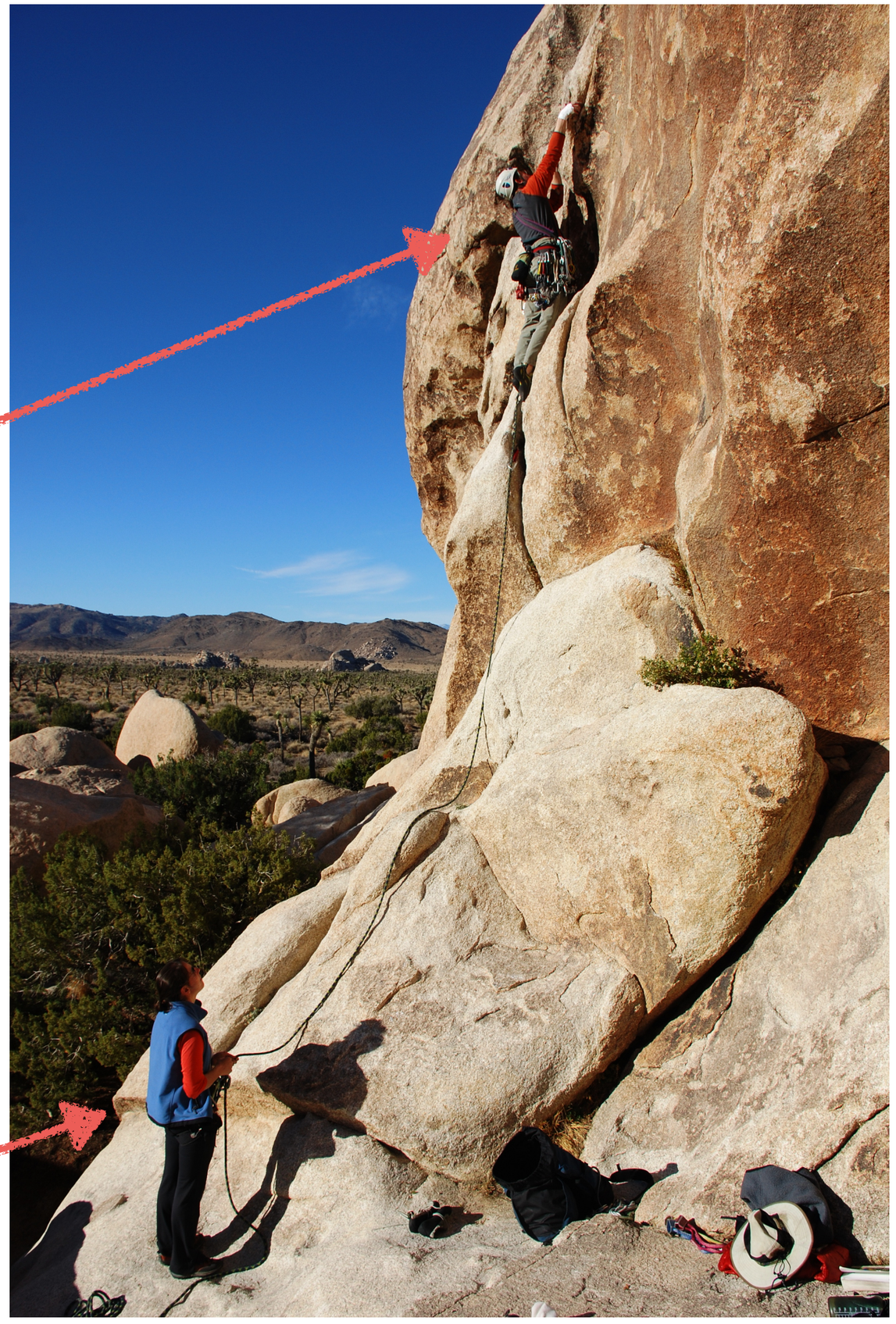
The transaction schedule that led to this log is ***unrecoverable*** (T1 reads data written by T2 but commits *before* T2).

Undo logging

- The main drawback of undo logging is the FORCE requirement: DB changes must be flushed to disk before commit.

Redo Logging

log



DB

Redo logging protocol

First two same as undo logging.

1. For each DB update, generate log record
2. Write ahead logging: before OUT(X), flush log records up to and including modifications of X.
3. Before T_i commits, flush log
4. No steal: before OUT(X), must write <T_i, commit> to log

Redo recovery protocol

1. Let winners be transactions with commit in log
2. For each log record from *first* to *last*:

   - If record was update
     `<`T_i`, A, old → new >`
     and T_i is winner, then X = new, W(X), OUT(X)

Intuition: repeat history for "winning" transactions.

Redo logging

Instructions: ~1 minute to think/ answer on your own; then discuss with neighbors; then I will call on one of you

Suppose a crash occurs and the log and DB are as shown. Use the redo recovery protocol to restore the DB.

(Challenge) Do the log records contain extra information that is never used during recovery? *If so, what is extra?*

Log

```
<T1 start>
<T2 start>
<T3 start>
<T3 B 8→12>
<T1 A 8→16>
<T2 A 16→32>
<T1 B 12→18>
<T3 commit>
```

DB

```
A: 8
B: 8
```