• Isolation
  • Conflict serializability
  • Conflict graphs
  • Two-phase locking
goal: build reliable systems from unreliable components
the abstraction that makes that easier is

transactions, which provide atomicity and isolation, while not hindering performance

atomicity  \rightarrow \text{shadow copies} (simple, poor performance) or logs (better performance, a bit more complex)

isolation  \rightarrow \text{two-phase locking}

eventually, we also want transaction-based systems to be distributed: to run across multiple machines
**goal:** run transactions $T_1$, $T_2$, $\ldots$, $T_N$ concurrently, and have it “appear” as if they ran sequentially

$T_1$
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit

t2
begin
write(x, 20)
write(y, 30)
commit

**naive approach:** actually run them sequentially, via (perhaps) a single global lock
goal: run transactions $T_1$, $T_2$, .., $T_N$ concurrently, and have it “appear” as if they ran sequentially

what does this even mean?

$T_1$
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit

$T_2$
begin
write(x, 20)
write(y, 30)
commit
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit

T2
begin
write(x, 20)
write(y, 30)
commit

possible sequential schedules
T1  ->  T2: x=20, y=30
T2  ->  T1: x=20, y=40

T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)

at end:
x = 20, y=40

T1: read(x)
T2: write(x, 20)
T1: tmp = read(y)
T2: write(y, 30)
T1: write(y, tmp+10)

at end:
x = 20, y=10
(assume x, y initialized to zero)
T1
begin
read(x)
tmp = read(y)
write(y, tmp+10)
commit

T2
begin
write(x, 20)
write(y, 30)
commit

possible sequential schedules

T1 -> T2: x=20, y=30
T2 -> T1: x=20, y=40

T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)

at end:
x = 20, y=40

T1: read(x)
T2: write(x, 20)
T1: tmp = read(y)
T2: write(y, 30)
T1: write(y, tmp+10)

at end:
x = 20, y=40
(assume x, y initialized to zero)
In the second schedule, T1 reads x=0 and y=30; those reads aren’t possible in a sequential schedule. Is that okay?
it depends.

there are many ways for multiple transactions to “appear” to have been run in sequence; we say there are different notions of **serializability**. what type of serializability you want depends on what your application needs.
conflicts

two operations conflict if they operate on the same object and at least one of them is a write.

conflict serializability

a schedule is conflict serializable if the order of its conflicts (the order in which the conflicting operations occur) is the same as the order of conflicts in some sequential schedule.
T1 begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit

T2 begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit

conflicts

T1.1 read(x) and T2.1 write(x, 20)
T1.2 tmp = read(y) and T2.2 write(y, 30)
T1.3 write(y, tmp+10) and T2.2 write(y, 30)
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit

T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit

conflicts

T1.1 read(x)  ->  T2.1 write(x, 20)
T1.2 tmp = read(y)  ->  T2.2 write(y, 30)
T1.3 write(y, tmp+10)  ->  T2.2 write(y, 30)

(T1  ->  T2)
T1
begin
T1.1 read(x)
T1.2 tmp = read(y)
T1.3 write(y, tmp+10)
commit

T2
begin
T2.1 write(x, 20)
T2.2 write(y, 30)
commit

conflicts

T2.1 write(x, 20)  ->  T1.1 read(x)
T2.2 write(y, 30)  ->  T1.2 tmp = read(y)
T2.2 write(y, 30)  ->  T1.3 write(y, tmp+10)

(T2 -> T1)
### Conflicts

| T1.1, T2.1 |
| T1.2, T2.2 |
| T1.3, T2.2 |

### Conflict Order for Sequential Schedules

<table>
<thead>
<tr>
<th>Initial Schedule</th>
<th>T1.1  -&gt;  T2.1</th>
<th>T2.1  -&gt;  T1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1.2  -&gt;  T2.2</td>
<td>T1.2  -&gt;  T2.2</td>
<td>T2.2  -&gt;  T1.2</td>
</tr>
<tr>
<td>T1.3  -&gt;  T2.2</td>
<td>T2.2  -&gt;  T1.3</td>
<td></td>
</tr>
</tbody>
</table>

(T1 -> T2)  
(T2 -> T1)

---

**T2.1**: write(x, 20)  
**T1.1**: read(x)  
**T2.2**: write(y, 30)  
**T1.2**: tmp = read(y)  
**T1.3**: write(y, tmp+10)

**T1.1**: read(x)  
**T2.1**: write(x, 20)  
**T2.2**: write(y, 30)  
**T1.2**: tmp = read(y)  
**T1.3**: write(y, tmp+10)

---

T2.1  ->  T1.1  
T2.2  ->  T1.2  
T2.2  ->  T1.3
conflicts

conflict order for sequential schedules

T1.1, T2.1
T1.2, T2.2
T1.3, T2.2

T1.1 → T2.1
T1.2 → T2.2
T1.3 → T2.2
(T1 → T2)

T2.1: write(x, 20)
T1.1: read(x)
T2.2: write(y, 30)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)

T2.1: write(x, 20)
T2.1: write(y, 30)
T2.2: write(y, tmp+10)
T1.1: read(x)
T1.2: tmp = read(y)
T1.3: write(y, tmp+10)

T2.1 → T1.1
T2.2 → T1.2
T2.2 → T1.3

T1.1 -> T2.1
T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3

(T2 → T1)
conflict graph

edge between $T_i$ and $T_j$ iff $T_i$ and $T_j$ have a conflict between them and the first step in the conflict occurs in $T_i$

$T_2$: write($x$, 20)
$T_1$: read($x$)
$T_2$: write($y$, 30)
$T_1$: tmp = read($y$)
$T_1$: write($y$, tmp+10)

T2 → T1

$T_1$: read($x$)
$T_2$: write($x$, 20)
$T_2$: write($y$, 30)
$T_1$: tmp = read($y$)
$T_1$: write($y$, tmp+10)

T2 ↔ T1

a schedule is conflict serializable iff it has an acyclic conflict graph
**Problem:** How do we generate schedules that are conflict serializable? Generate all possible schedules and check their conflict graphs?
solution: two-phase locking (2PL)

1. each shared variable has a lock

2. before any operation on a variable, the transaction must acquire the corresponding lock

3. after a transaction releases a lock, it may not acquire any other locks
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire((x).lock)</td>
<td>acquire((y).lock)</td>
</tr>
<tr>
<td>read((x))</td>
<td>read((y))</td>
</tr>
<tr>
<td>acquire((y).lock)</td>
<td>acquire((x).lock)</td>
</tr>
<tr>
<td>read((y))</td>
<td>read((x))</td>
</tr>
<tr>
<td>release((y).lock)</td>
<td>release((x).lock)</td>
</tr>
<tr>
<td>release((x).lock)</td>
<td>release((y).lock)</td>
</tr>
</tbody>
</table>

**Problem:** 2PL can result in deadlock
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire(x.lock)</td>
<td>acquire(y.lock)</td>
</tr>
<tr>
<td>read(x)</td>
<td>read(y)</td>
</tr>
<tr>
<td>acquire(y.lock)</td>
<td>acquire(x.lock)</td>
</tr>
<tr>
<td>read(y)</td>
<td>read(x)</td>
</tr>
<tr>
<td>release(y.lock)</td>
<td>release(x.lock)</td>
</tr>
<tr>
<td>release(x.lock)</td>
<td>release(y.lock)</td>
</tr>
</tbody>
</table>

**solution:** global ordering on locks
**T1**
- acquire(`x.lock`)
- read(`x`)
- acquire(`y.lock`)
- read(`y`)
- release(`y.lock`)
- release(`x.lock`)

**T2**
- acquire(`y.lock`)
- read(`y`)
- acquire(`x.lock`)
- read(`x`)
- release(`x.lock`)
- release(`y.lock`)

**better solution:** take advantage of atomicity and abort one of the transactions!
**performance improvement:** allow concurrent reads with reader- and writer-locks

can hold reader-locks at the same time as other readers

can release reader-locks after transaction reaches lock point (which may be before commit)
Different types of **serializability** allow us to specify precisely what we want when we run transactions in parallel. **Conflict-serializability** is common in practice.

**Two-phase locking** allows us to generate conflict serializable schedules. We can improve its performance by allowing concurrent reads via reader- and writer-locks.