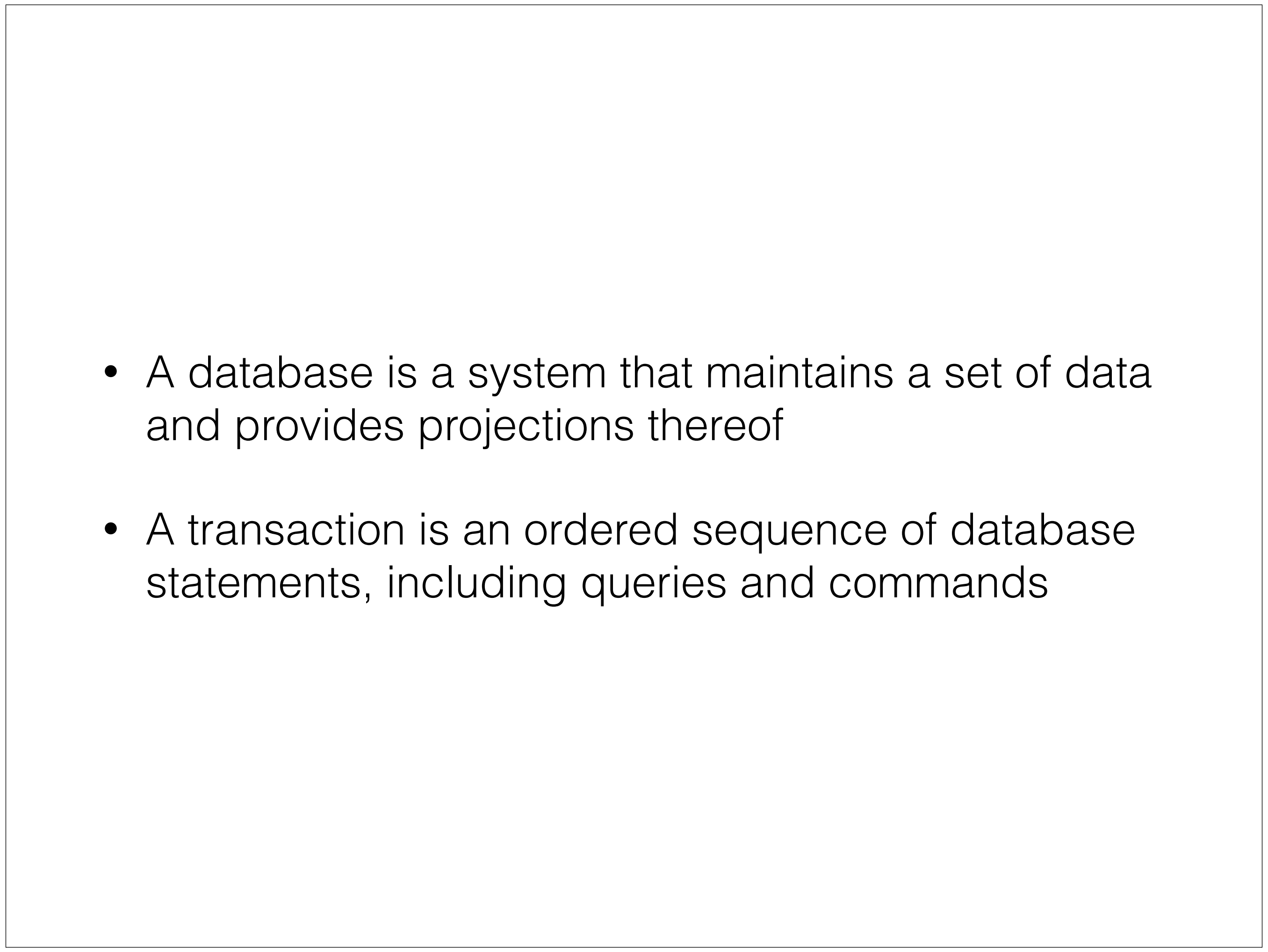
MySQL and ACID

Donald Ball

Hi everybody! A couple of months ago I was struggling with some database deadlock issues I couldn't reason out and spent a vaguely enjoyable afternoon reading the MySQL documentation cover to cover. To validate my understanding, and to perhaps spare others from this quest (alternately, to gain companions in my ongoing quest), I offered to give a talk on what I (think) I've learned.

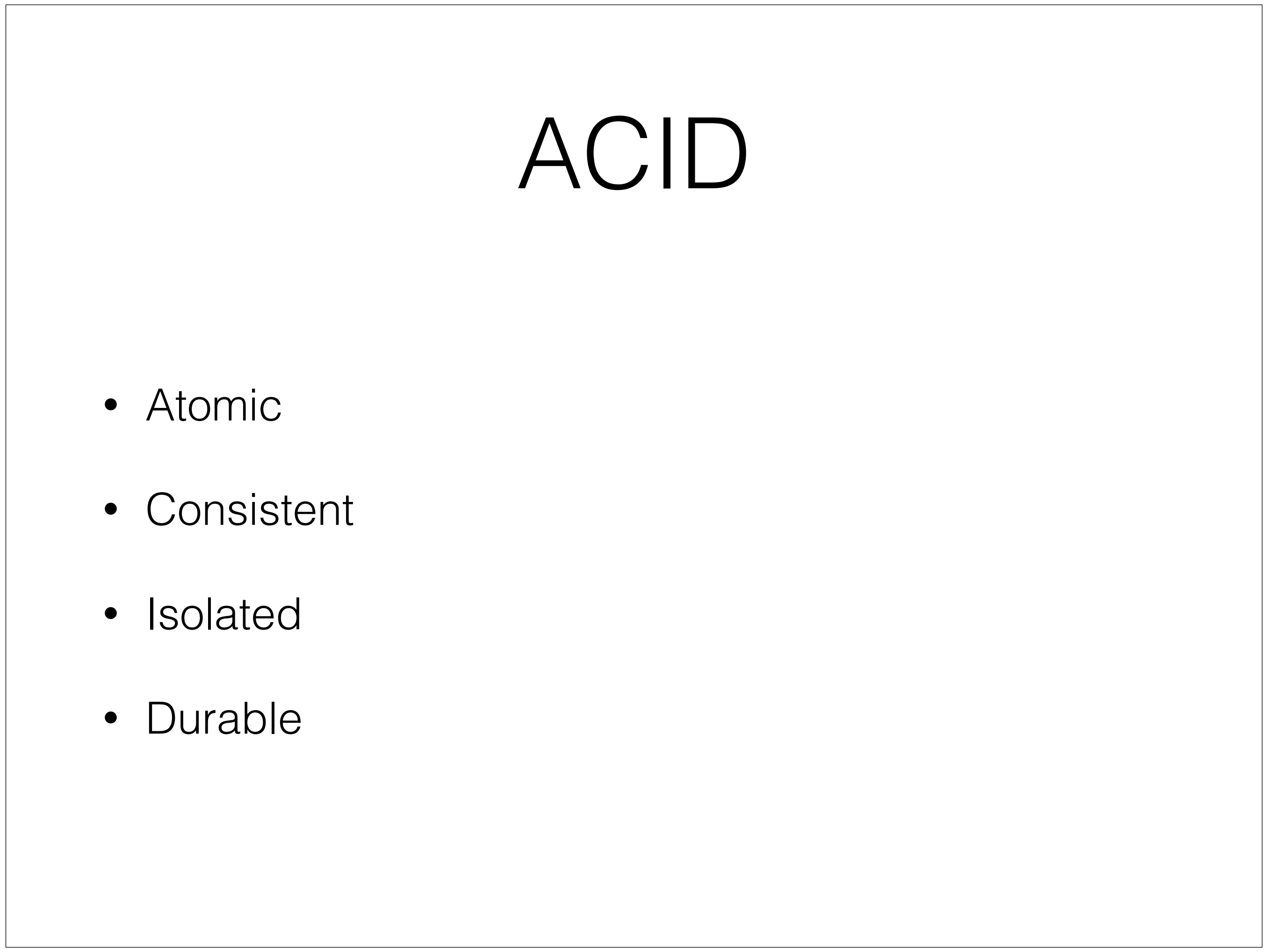
I should emphasize that I'm not a database guru, no one has reviewed this talk for technical content, so if you sense I'm lying to you, it's entirely possible I am; please correct me.

I should note that when I refer to MySQL in this talk, I really mean the MySQL InnoDB engine. If you're using MyISAM, you may want to take a walk outside for the next 10 minutes because you obviously have no regard for the safety of your data (or you understand these tradeoffs better than I and should be giving a subsequent lightning talk).



- A database is a system that maintains a set of data and provides projections thereof
- A transaction is an ordered sequence of database statements, including queries and commands

Let’s define a couple of terms to get warmed up. A database is a system that maintains a set of data and provides projections thereof. Typically, database offer commands to change data and queries to evaluate projections. A transaction is an ordered sequence of database statements, including both queries and commands.



ACID

- Atomic
- Consistent
- Isolated
- Durable

Many databases, particularly relational SQL databases, offer a set of guarantees colloquially termed, “ACID”.
Atomicity guarantees that either all commands of a transaction succeed, or none do. This is a requirement when performing coordinated data changes, e.g. debiting and crediting account summary balances.
Consistency means that the data are consistent with the rules set forth in the database schema. For example, that there is a table named “deals” wherein is found the truth of all things. Relational schemas tend to offer fairly modest consistency contracts, generally describing tables’ fields’ types, the uniqueness of the values of various combinations of fields, and possibly some guarantees about the existence of referenced entities.
Isolation means that concurrent transitions are processed such that the subsequent database state is the result of a serial ordering of the successful transactions. In order to provide this guarantee, database transactions may block or deadlock.
Durability means that a committed transaction may not be lost. This is a useful property for a database to have.

Isolation

Of these properties, isolation is the most subtle and mysterious. Let’s explore!

MySQL guarantees isolation using MVCC, multi-version concurrency control. In a nutshell, this means it keeps track of the portions of the database each transaction reads or writes. If another transaction wants to read or write the same portions concurrently, it may join the lock (e.g. concurrent reads), block (waiting until the locks are released), or deadlock (if it holds a lock on which the other transaction is blocked).

Of course, this raises more questions: what kind of locks are there, and what do we mean by “portions of the database”?

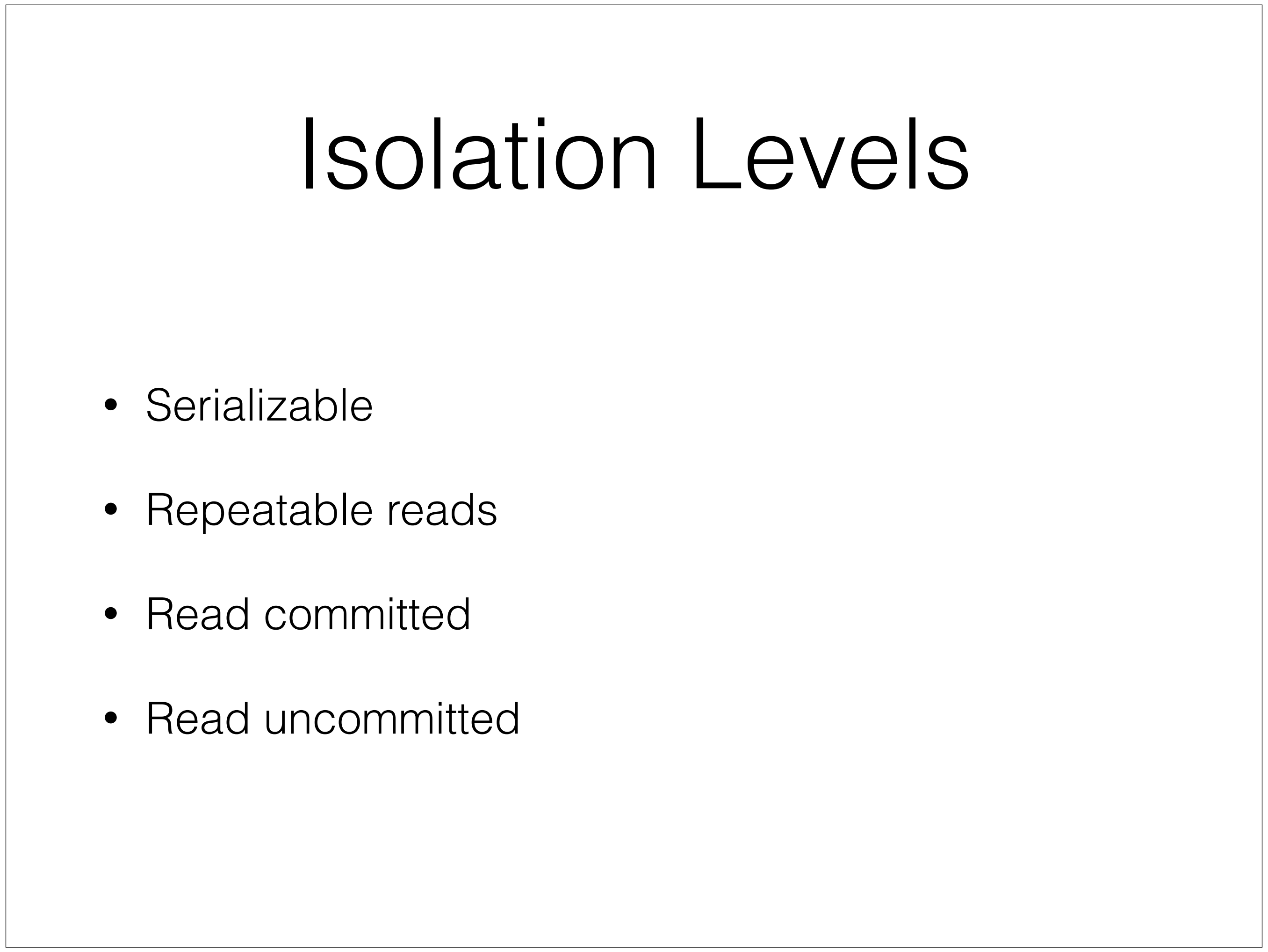
- Updates, deletes, and write-locking selects obtain write locks
- Selects obtain no locks, read locks, or write locks depending on the isolation level
- Row locks aren’t actually obtained on rows; they’re obtained on the index values scanned in a query
- Inserts also obtain locks on the gap preceding the index values

In MySQL: updates, deletes, and write-locking selects obtain write locks. Read-locking selects obtain read locks.

This is the key bit that eluded me for quite some time: row locks aren’t obtained on rows, they’re obtained on index values. (Names have power. In this case, the power to mislead.) “Row locks” are obtained on the index values scanned in the query.

Inserts are a little weird. They also obtain locks on the gap, if any, between the index value of the inserted row and the preceding index value.

There are also intention locks, but I don’t even pretend to really understand them yet, so we’re going to move on. This is just a lightning talk.



Isolation Levels

- Serializable
- Repeatable reads
- Read committed
- Read uncommitted

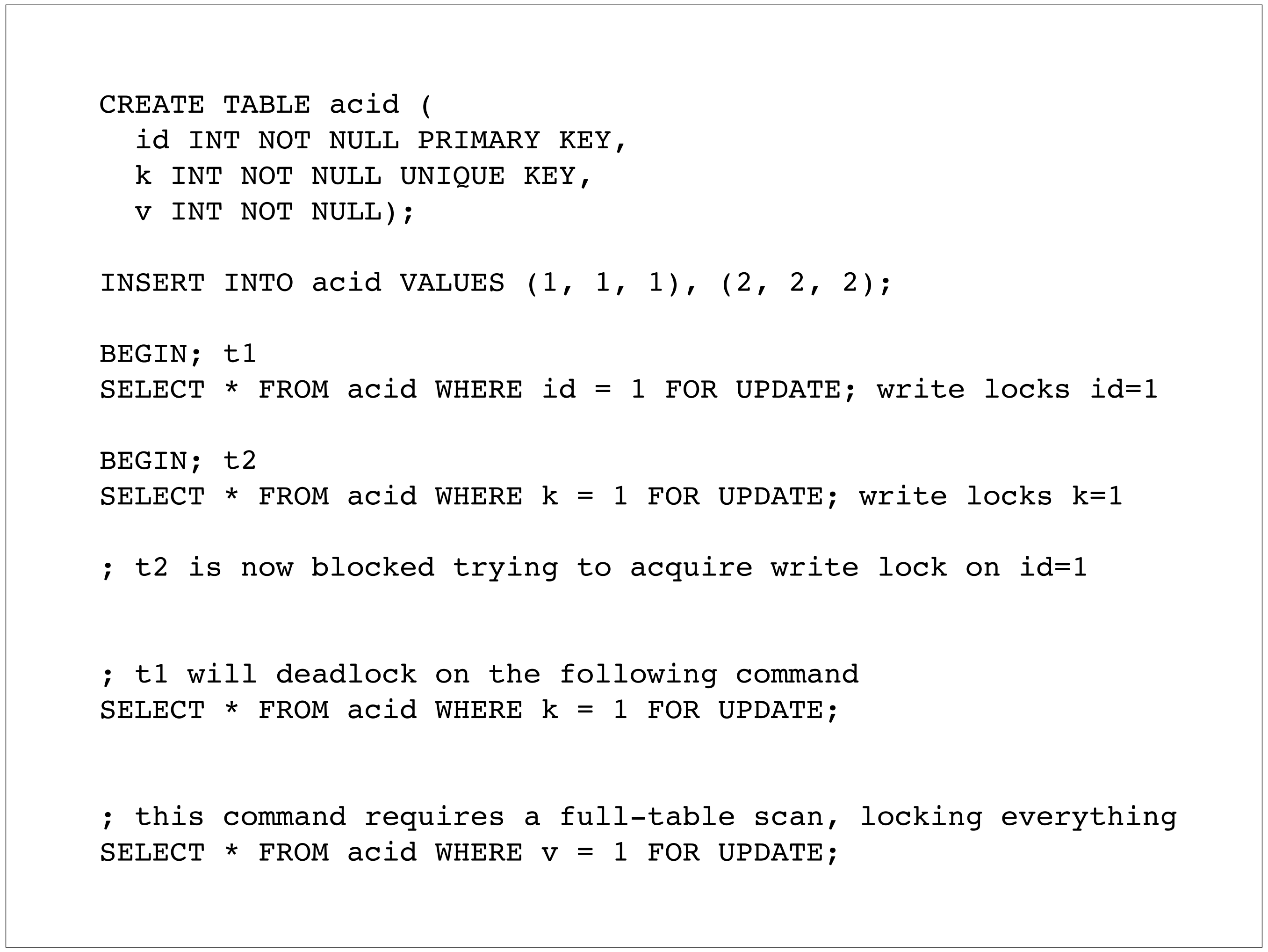
In a perfect world, transactions would be effected instantaneously and they would be fully linearized. Also I would have a pony and a million dollars. Alas, we live in a fallen world, so I have no pony and linearized transactions are too slow for many cases. SQL provides four ostensibly well-defined isolation levels that allow users to choose between speed and data stability.

Serializable transactions guarantee isolation as defined earlier. MySQL implements this by implicitly changing all SELECT commands to obtain write locks, as if they were given the FOR UPDATE condition.

Repeatable reads is the default isolation level. In it, reads always return the same rows, but other transactions are not blocked from changing them. This means that a transaction may decide to make changes based on a projection of the database that is no longer true.

Read committed: Reads are not consistent within the same transaction. I suppose this is useful if what you really care about is atomic rollback, but it seems ill-advised.

Read uncommitted: Reads see uncommitted results from concurrent transactions. Um. Ok. Hey, if you want to observe a state of the database that may never have been true, knock yourself out.

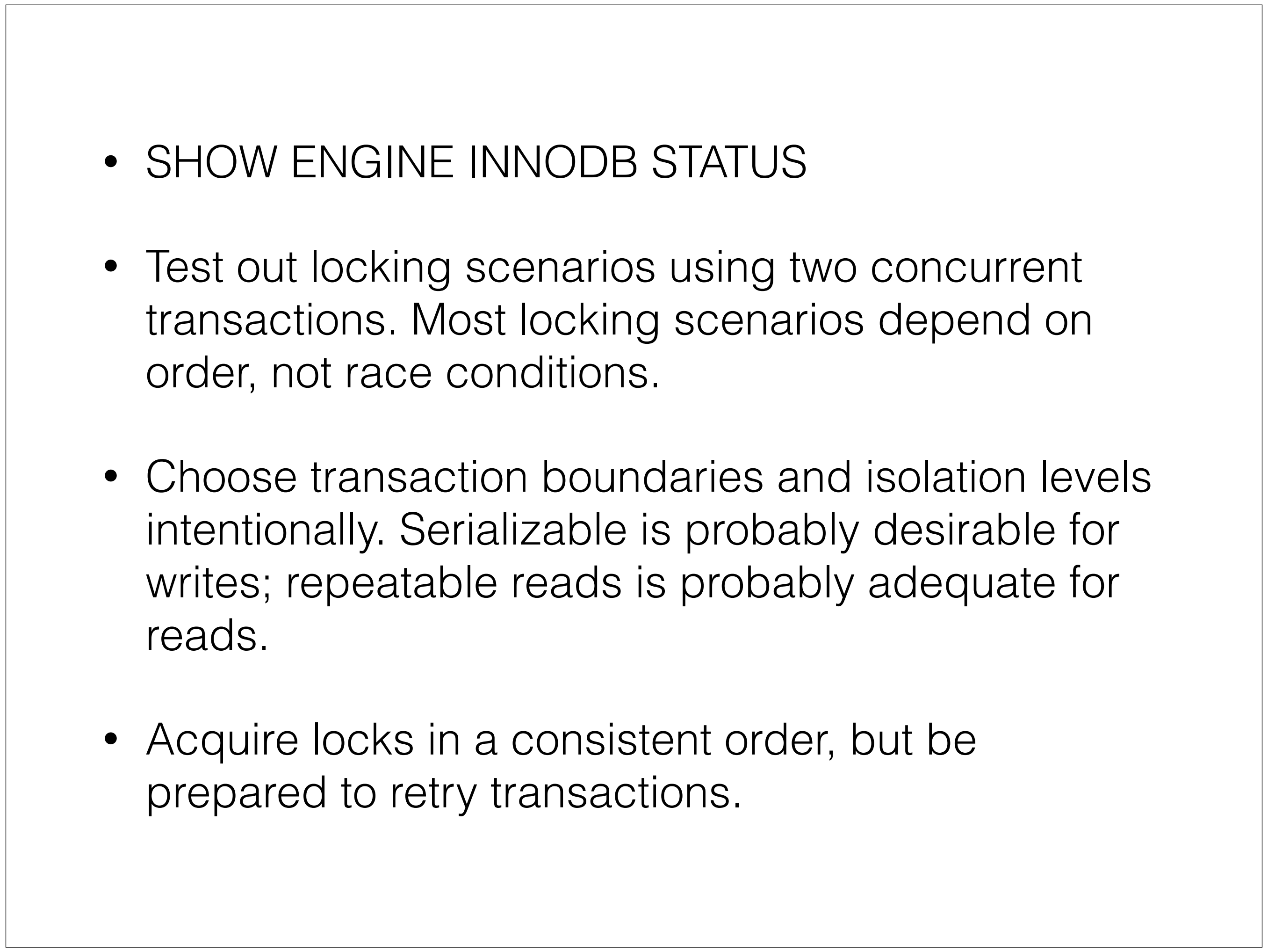


```
CREATE TABLE acid (
  id INT NOT NULL PRIMARY KEY,
  k INT NOT NULL UNIQUE KEY,
  v INT NOT NULL);

INSERT INTO acid VALUES (1, 1, 1), (2, 2, 2);

BEGIN; t1
SELECT * FROM acid WHERE id = 1 FOR UPDATE; write locks id=1

BEGIN; t2
SELECT * FROM acid WHERE k = 1 FOR UPDATE; write locks k=1

; t2 is now blocked trying to acquire write lock on id=1


; t1 will deadlock on the following command
SELECT * FROM acid WHERE k = 1 FOR UPDATE;


; this command requires a full-table scan, locking everything
SELECT * FROM acid WHERE v = 1 FOR UPDATE;
```

So how does all this stuff really work? I find it's easiest to reason about if I play with a simple table and two concurrent transactions. Here I create a table named "acid" with a primary key, a unique key, and a normal field. I then explore what happens in a simple concurrent scenario. Transaction 1 issues a write-locking select on the table by primary key and obtains a write lock on the primary key with value 1. Transaction 2 then issues a write-locking select on the table by unique key. T2 is able to obtain a write lock on the unique key with value 1, but then blocks trying to obtain a write lock on the primary key with value 1.

Here's where the naive belief that transactions lock rows breaks down. If T1 then issues a write-locking select on the table by unique key - for the row that it has ostensibly locked already - the transactions deadlock due to the circular lock references.

Another interesting takeaway is that the final query acquires a write-lock on all primary key index values. There is no index on v, so it has to do a full-table scan, and MySQL locks all index values it scans.



- SHOW ENGINE INNODB STATUS
- Test out locking scenarios using two concurrent transactions. Most locking scenarios depend on order, not race conditions.
- Choose transaction boundaries and isolation levels intentionally. Serializable is probably desirable for writes; repeatable reads is probably adequate for reads.
- Acquire locks in a consistent order, but be prepared to retry transactions.

If you choose to explore these matters more deeply, the command "SHOW ENGINE INNODB STATUS" will be your friend. It tells you the locks held by the active transactions and remembers the details of the most recent deadlock.

Test out locking scenarios using two concurrent transactions. Most database locking scenarios depend on the order of commands, not race conditions that are hard to reproduce. Increase the MySQL lock timeout if you need more time to think interactively.

Choose transaction boundaries and isolation levels intentionally. Serializable is probably desirable for writes; repeatable reads is probably adequate for reads.

If you can, acquire your locks in a consistent order, both across tables and within tables, to avoid circular lock references, but always be prepared to retry transactions; some schemas make deadlocks unavoidable and lock timeouts are always a possibility. Keep transactions as short as possible.

That's all I've got, thanks for listening.