Today:

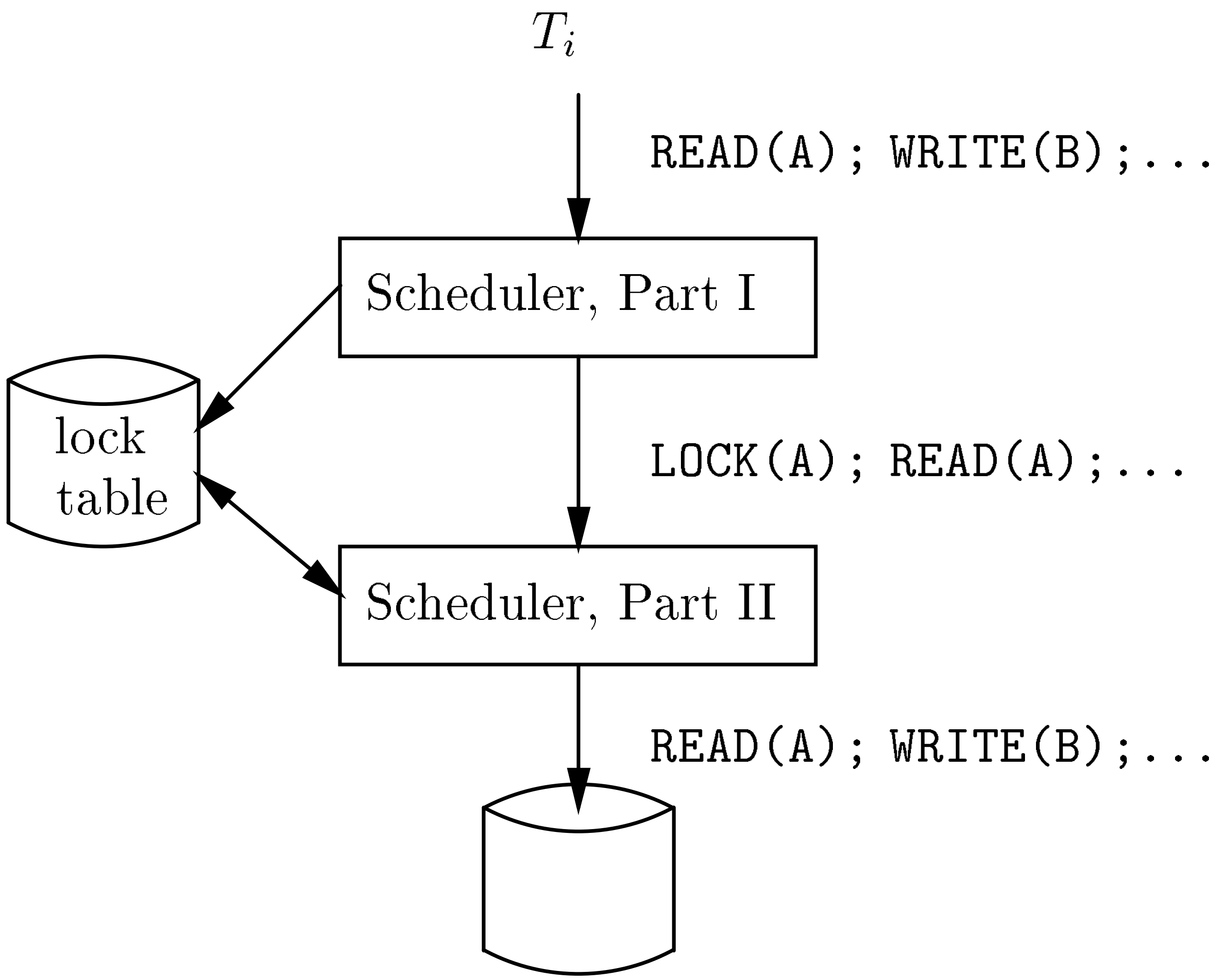
- The scheduler.
- Some different kinds of lock modes and their compatability matrices.
  - Intension locks (multigranularity locks), tree protocol.
- What is a lock table, anyway?

Soon:

- Increment locks.
- View-serializability: serializable schedules that are not conflict-serializable.
- Other ways of maintaining serializability.
- Deadlocks.

A Simple Lock Manager/Scheduler

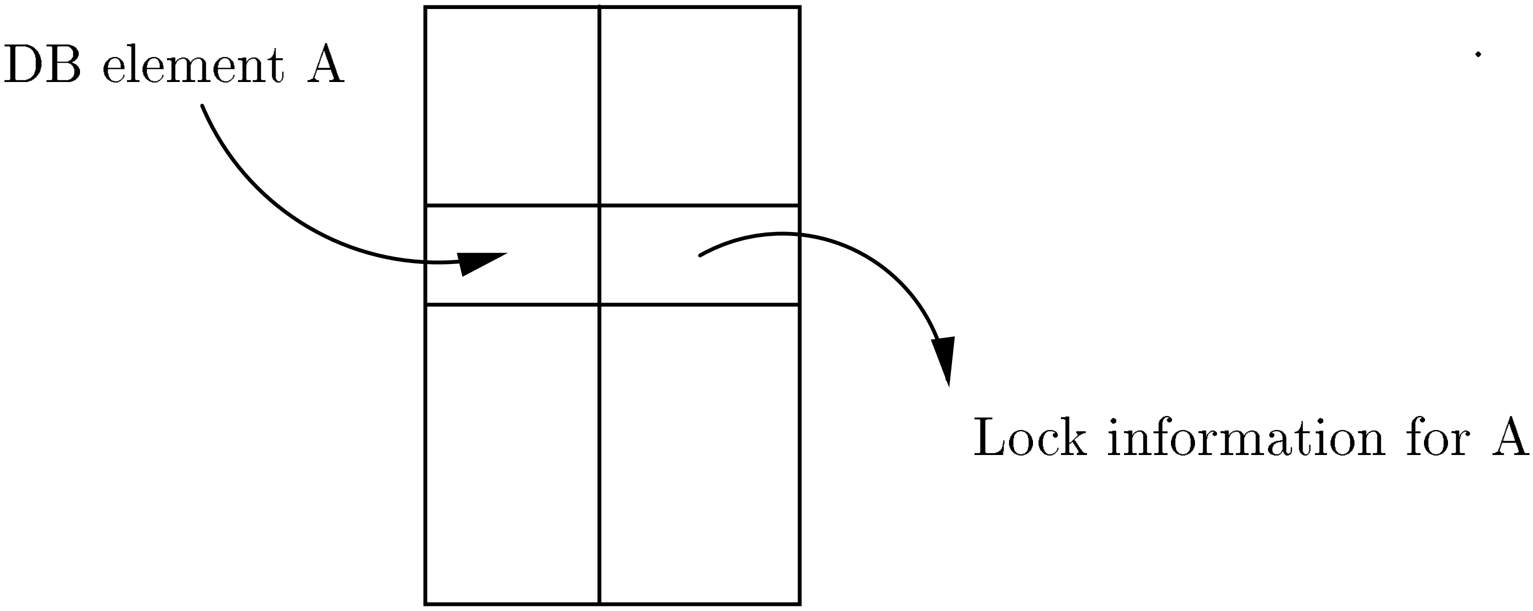
- Don't trust the transactions to request locks.
- Hold all locks until `COMMIT` or `ABORT`.



- First part of scheduler inserts locks in front of actions.
- Second part of scheduler manages locks; only transmits requests to the database when the lock table contents, the rules for granting new locks, and the legal-schedule rules allow it.
- First part issues unlocks to lock table after transaction commit.

Contents of the Lock Table

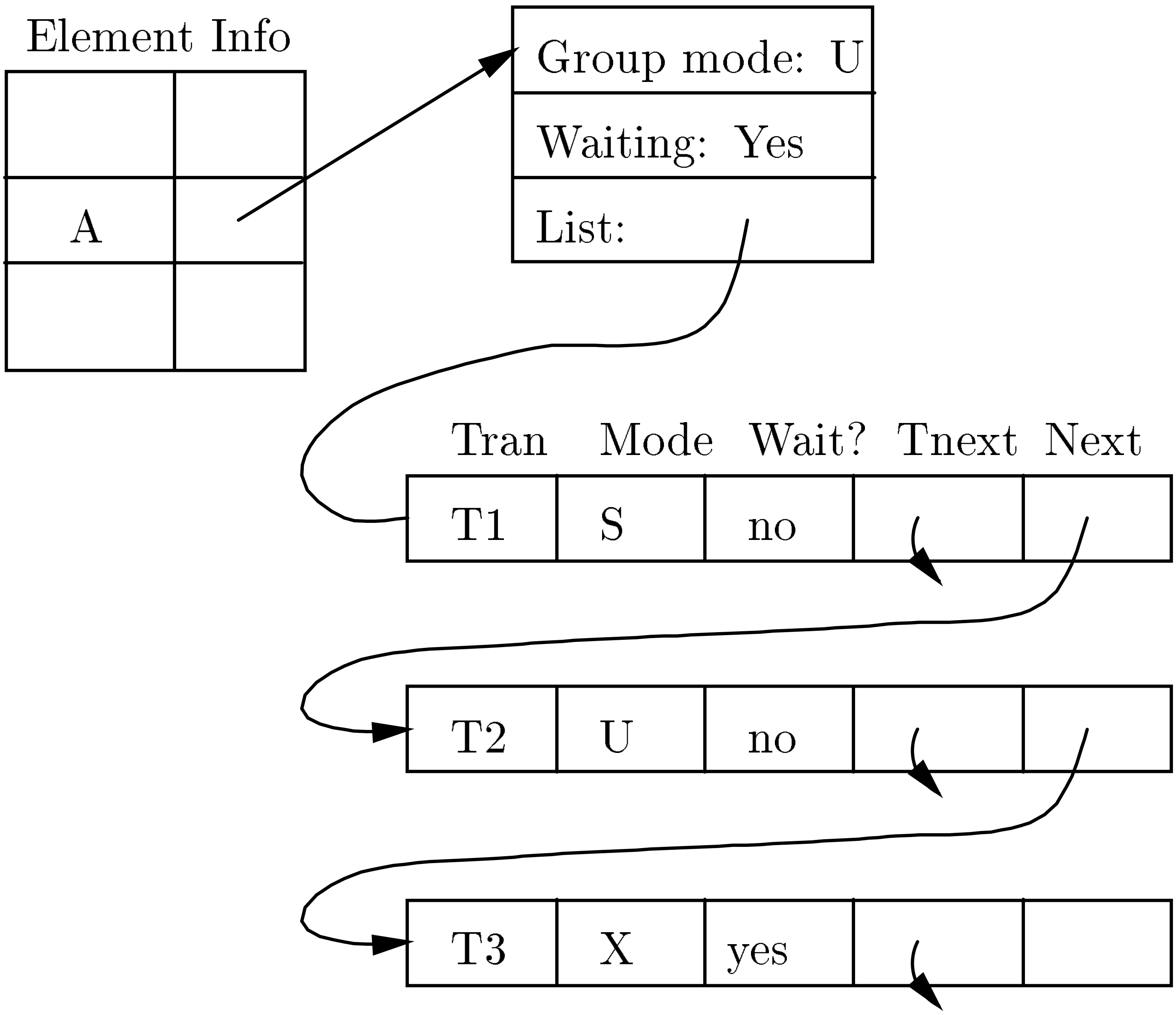
Abstractly, a mapping:



- Implement with hash table.
    - ❖ Not present = not locked.

Lock Information for a DB Element

- List of all transactions currently holding, or waiting for, locks.
- Summary ("group mode") to help determine whether other locks can be granted.



Granularity of Locks

"DB element" is a purposely vague term. Some options:

1. Relations.
2. Tuples.
3. Blocks/pages.

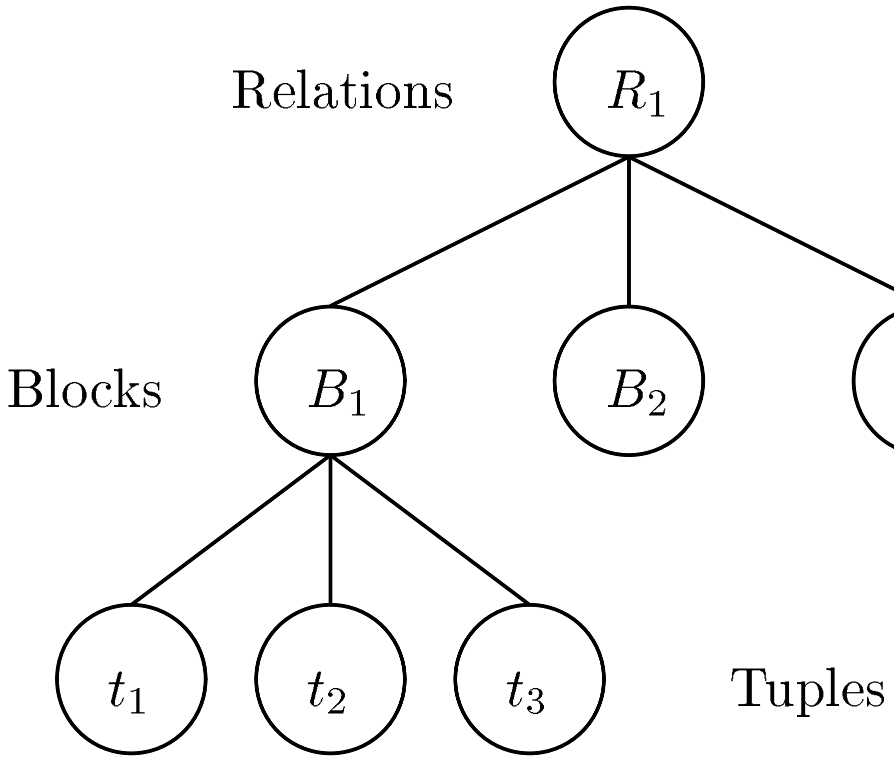
Tradeoff

- Large elements = little concurrency; small elements = more concurrency.
- Large elements = easy lock management; small elements = large lock table.
- Example: consider a bank's database. Could you have just one lockable element: the accounts relation?
- Another example: a DB of documents. Would you want to lock anything smaller than a complete document?
  - The fact that almost all transactions are read-only weakens the need for fine-grained locks.

Having it Both Ways

A system can support multiple-granularity locks.

- Problem: if I lock a relation, and you lock one of its tuples, can we indulge in unserializable behavior?
- Solution: think of DB elements as a hierarchy. DB composed of relations, composed of blocks, composed of tuples.



Warning Protocol

- Start at root.
- If you want to read/write an element, put a lock on it.
- If you want to read/write a subelement, put a *warning* ("intention," denoted I) on it, and proceed to the appropriate child or children.
- Locking rule: You cannot put a lock where someone else has a warning, but you can also put a warning.

Example

I want to lock tuple t_1 of relation R; you want to lock tuple t_2.

- We both put a warning on R.
- We each put a warning on the block containing our desired tuples (may or may not be the same block).
- We each lock our respective tuples.

Caveat: Phantoms

We can only lock things that exist, not those things that might exist.

- Example: I want to compute:

```
SELECT SUM(length)
FROM Movies
WHERE studioName = 'Disney';
```

- I put a warning on `Movies` and a lock on all tuples for Disney movies.
- At the same time, you insert a new Disney movie into a block I did not lock.
  - Could result in DB inconsistency, if my sum gets into the database somewhere.

Solution

DB modifications to a relation should be treated as write actions on the relation as a DB element.

- You must get a lock, not just a warning, on `Movies`, in order to insert a new Disney movie.
  - Blocked by my warning on `Movies`.

A Shared/Exclusive/Warning Scheme

If we want to distinguish S and X locks as well as warnings, we get 5 different lock modes and the following compatibility matrix.

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

A Special Protocol For Trees

- All prior locking schemes assume 2PL to work.
- 2PL is a disaster for something like a B-tree index.
  - ❖ Remember, any insert or delete might wind up rewriting the root.
  - ❖ Every DB modification using the B-tree would have to lock the root (at least), so only one such transaction at a time could operate.
- Most of the time, we can tell immediately that a node will not have to be rewritten.
  - ❖ Example: We are doing an insert, and the child of the root that we visit is not full. There is no way we will have to split that node and rewrite the root.

Rules for Locking Tree Nodes

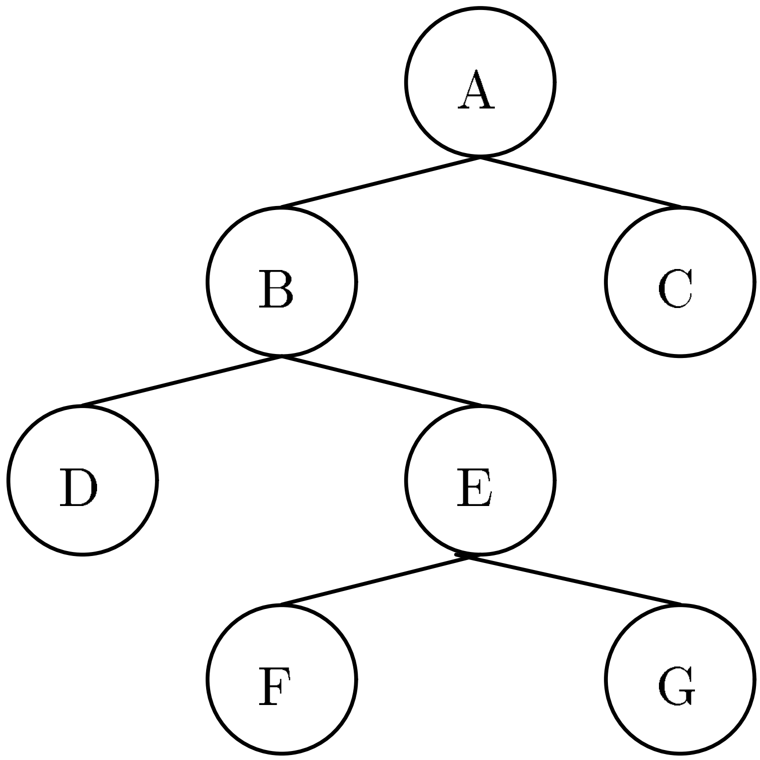
1. Start by locking any node (B-tree application always starts at root).

2. Subsequent nodes can only be locked if you have a lock on its parent.

3. Nodes may be unlocked at any time.

4. You cannot relock an item you previously unlocked, even if you still hold a lock on its parent.

Example



T_1	T_2	T_3
$l_1(A);\ r_1(A);$		
$l_1(B);\ r_1(B);$		
$l_1(D);\ r_1(D);$		
$w_1(B);\ u_1(B);$		
	$l_2(B);\ r_2(B);$	
		$l_3(E);\ r_3(E);$
$w_1(D);\ u_1(D);$		
		$l_3(F);\ r_3(F);$
$w_1(A);\ u_1(A);$		
		$w_3(E);\ u_3(E);$
	$l_2(E);\ r_2(E);$	
		$w_3(F);\ u_3(F);$
	$w_2(B);\ u_2(B);$	
	$w_2(E);\ u_2(E);$	

Why It Works

- If all transactions start at the root, as for B-trees, then no transaction can "pass" another going down the tree.
  - Because a transaction must hold at least one lock until it is finished locking.
- Even if transactions start at nonroots, they can never pass each other.
  - Formally: if the highest node that T_1 and T_2 lock in common is X, and T_1 locks X first, then every node that they lock in common will be locked first by T_1.