

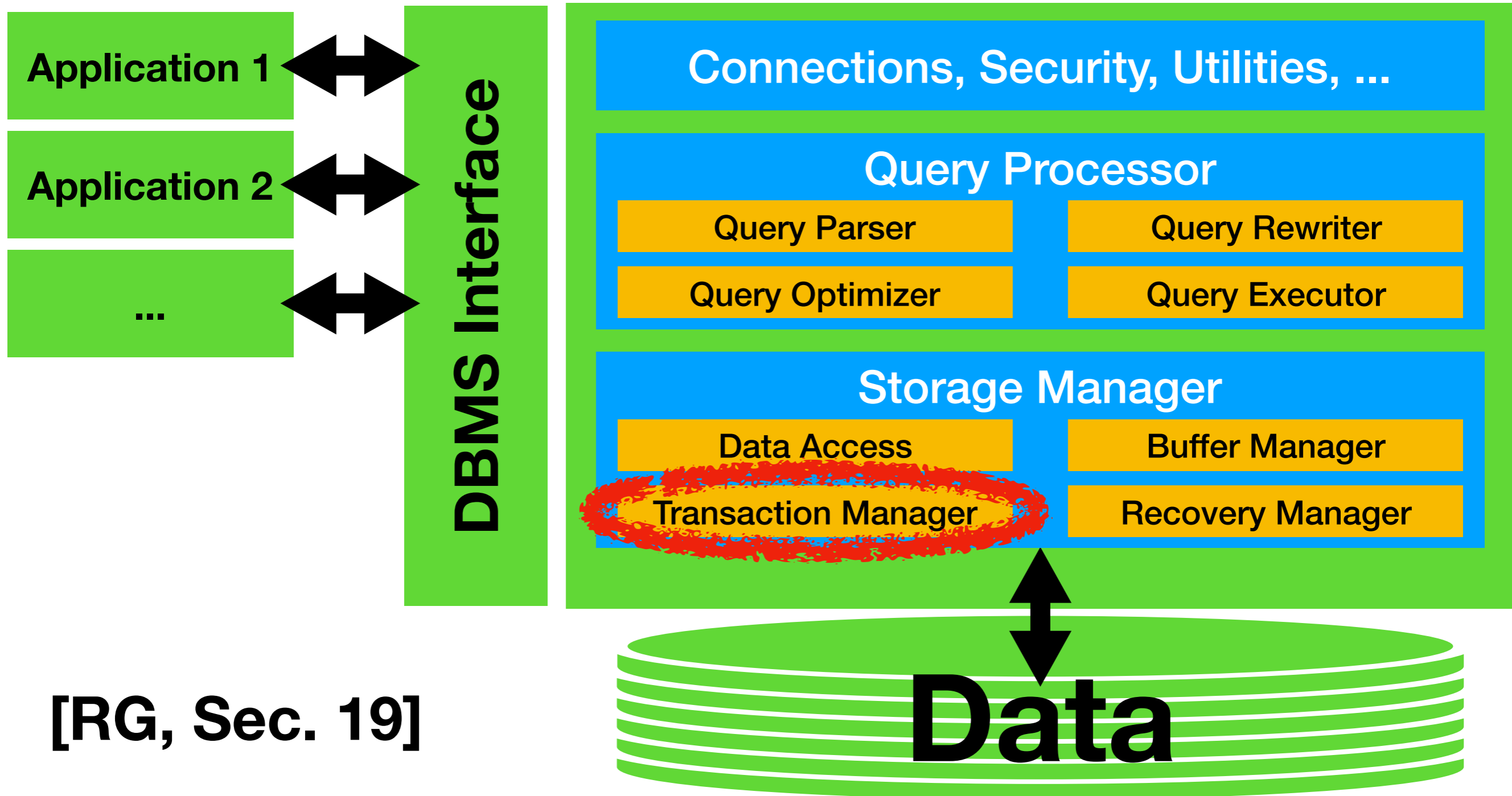
More on Locking

Immanuel Trummer

itrummer@cornell.edu

www.itrummer.org

Database Management Systems (DBMS)



[RG, Sec. 19]

Outlook

- Handling **deadlocks**
- Handling **phantoms**
- Efficient **index** locking
- **Multi**-granularity locking

Deadlocks

Handling Deadlocks

- **Deadlocks** can arise when using non-conservative 2PL
 - Deadlock: **transactions waiting** in a "circle"
 - May be acceptable if deadlocks are **rare**
- Two ways for **handling deadlocks**
 - **Detect** and resolve deadlocks
 - **Prevent** deadlocks from happening

***Task: Generate
Deadlock in Postgres!***

Deadlock Detection

- Simplest option: assume deadlock after **timeout**
- Maintain **waits-for graph** to detect deadlocks
 - One node for each **transaction**
 - Edge from T1 to T2 if T1 **waits for** lock held by T2
 - Edges are added as **lock requests** come in
 - **Cycle** in waits-for graph indicates a deadlock

Resolving Deadlocks

- Only possibility: **abort** one deadlocked transaction
- Aborted transaction is typically **restarted**
- Can try to **optimize** selection of aborted transaction
 - E.g., abort **youngest** transaction for least overhead

Avoiding Deadlocks

- **Proactively abort transactions** that may cause deadlocks
- Priority based **on timestamps** (older transaction - higher priority)
- Transaction T1 needs lock held by T2 - **Wound-wait protocol**:
 - T1 causes T2 abort if T1 has **higher** priority
 - T1 waits for lock from T2 if T1 has **lower** priority
- Transaction T1 needs lock held by T2 - **Wait-die protocol**:
 - T1 waits for lock from T2 if T1 has **higher** priority
 - T1 aborts itself if it has **lower** priority than T2

Wound Wait

Deadlock Prevention Proof

- A deadlock means transactions **wait in a cycle**
- Only **lower priority transaction can wait** for higher priority
 - Due to definition of wound-wait protocol
- **Assume cycle** in waits-for graph, transaction T1 in cycle
 - $T1 \rightarrow T2$: T1 must have **lower** priority than T2
 - $T1 \rightarrow T2 \rightarrow T3$: T1 must have **lower** priority than T3
 - $T1 \rightarrow \dots \rightarrow T1$: T1 must have **lower** priority than T1
 - Leads to a **contradiction** so no cycle is possible!

Wait-Die

Deadlock Prevention Proof

- A deadlock means transactions **wait in a cycle**
- Only **higher priority transaction can wait** for lower priority
 - Due to definition of wait-die protocol
- **Assume cycle** in waits-for graph, transaction T1 in cycle
 - $T1 \rightarrow T2$: T1 must have **higher** priority than T2
 - $T1 \rightarrow T2 \rightarrow T3$: T1 must have **higher** priority than T3
 - $T1 \rightarrow \dots \rightarrow T1$: T1 must have **higher** priority than T1
 - Leads to a **contradiction** so no cycle is possible!

Wound-Wait vs. Wait-Die

- **Advantage** of Wait-Die:
 - Transactions that acquired all locks won't abort
- **Disadvantage** of Wait-Die:
 - Young transaction may re-abort for same reason

Avoiding Starvation

- Higher priority transaction is **never restarted** for both
- When restarting transaction, assign **original timestamp**
- So transaction will be **eventually prioritized**
- Avoids **starvation** (i.e., no transaction never processed)

Phantoms

Phantom Example

- Transaction 1 selects students with name **starting with F**
- Transaction 2 inserts new student "**Frank**"
- Transaction 1 selects students **starting with F again**
 - Suddenly we see **a new student** in the query result
 - Similar to **unrepeatable read**, caused by insertions
- Problem: 2PL only locked students present **at first query**

Avoiding Phantoms

- **Predicate locking**: lock tuples satisfying certain predicate
 - E.g., predicate "**name starts with F**" in the example
 - Locks **current and future** entries equally
 - **Complex** to realize for arbitrary predicates
- **Can use index** when considering equality predicates
 - **Lock index page** that would change at insertion
 - **Cannot insert** as long as index page is locked

Efficient Index Locking

*Tree Indexes: Why Not
Use Generic 2PL?*

Locking in Tree Indexes

- Observation: we traverse tree into **one direction** only
- **Locking one node** sufficient to block other transactions
 - I.e., keeping later transactions **out** of current sub-tree
- Locking for **index lookups** ("crabbing"):
 - Identify **next node** (child node or root at start)
 - Lock **next** (read lock), then **unlock** parent - repeat

Illustration of Crabbing

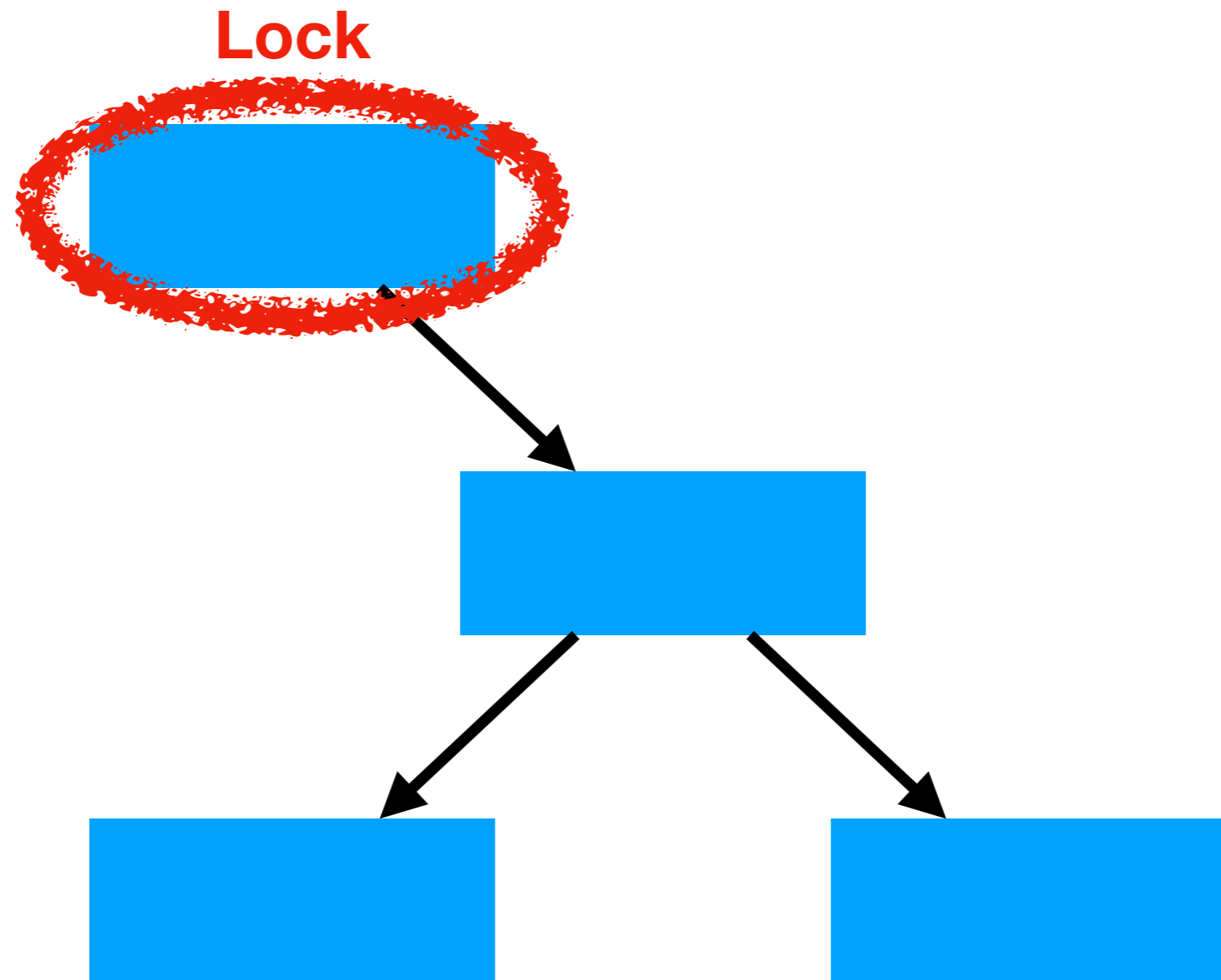


Illustration of Crabbing

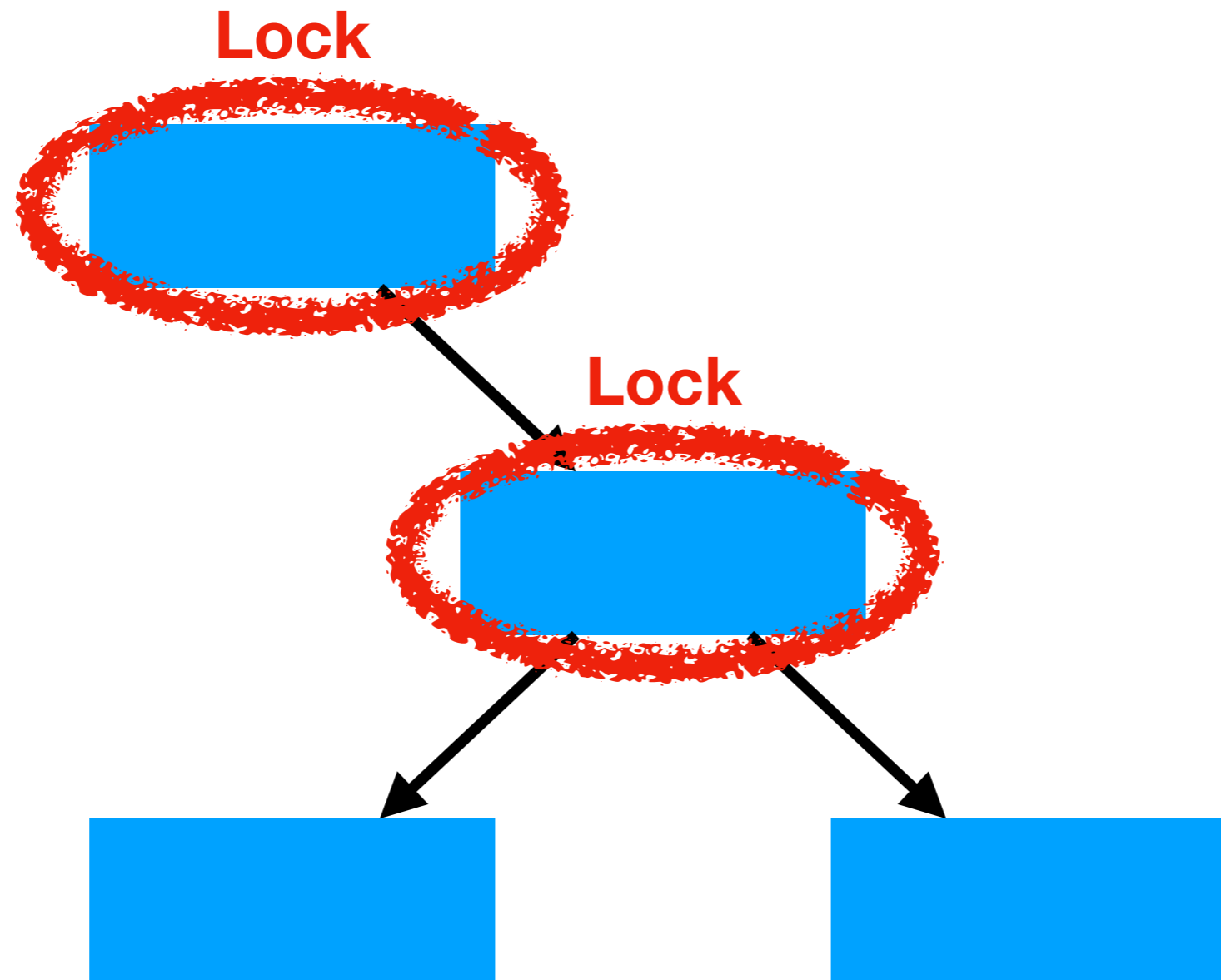


Illustration of Crabbing

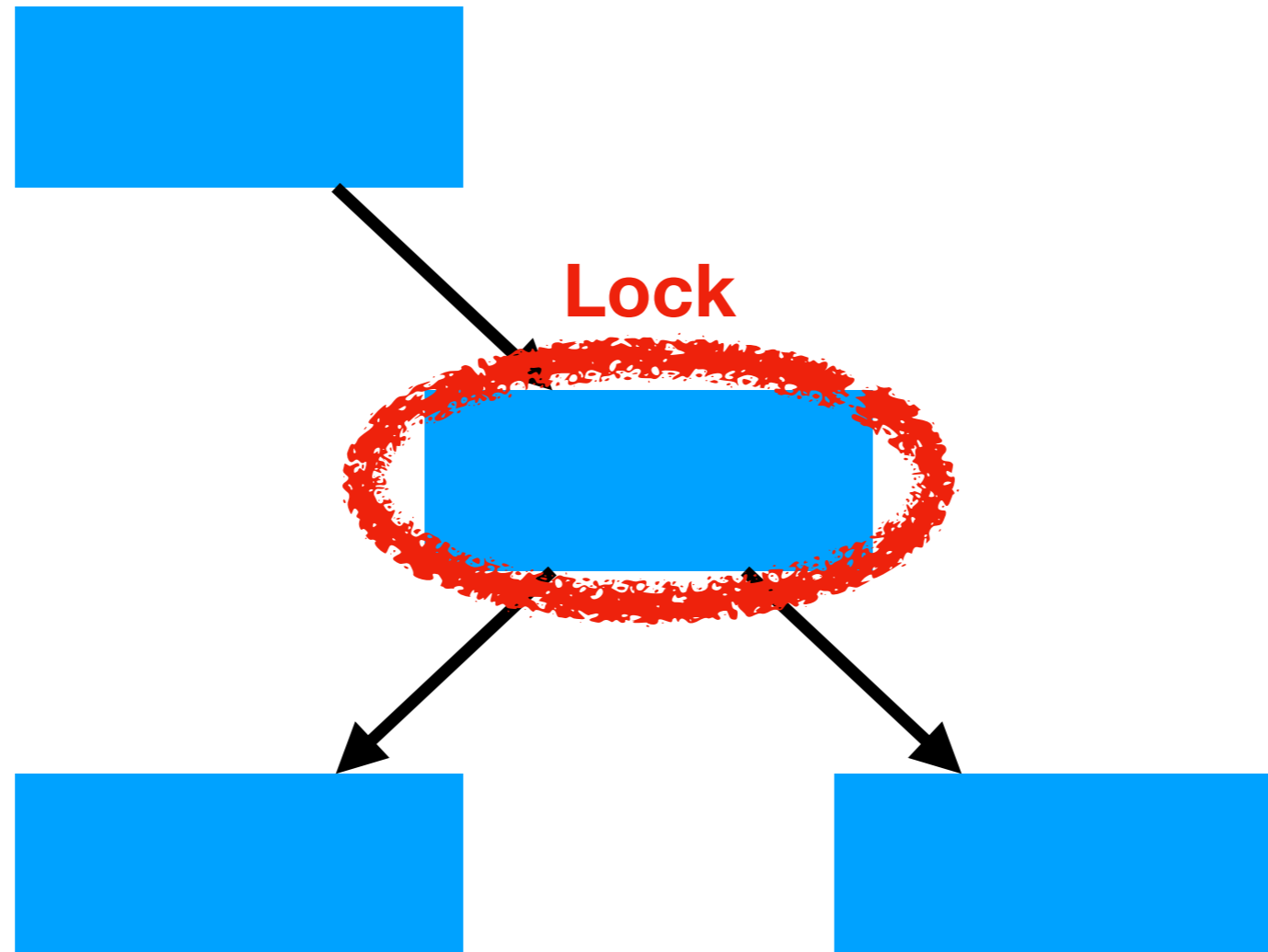


Illustration of Crabbing

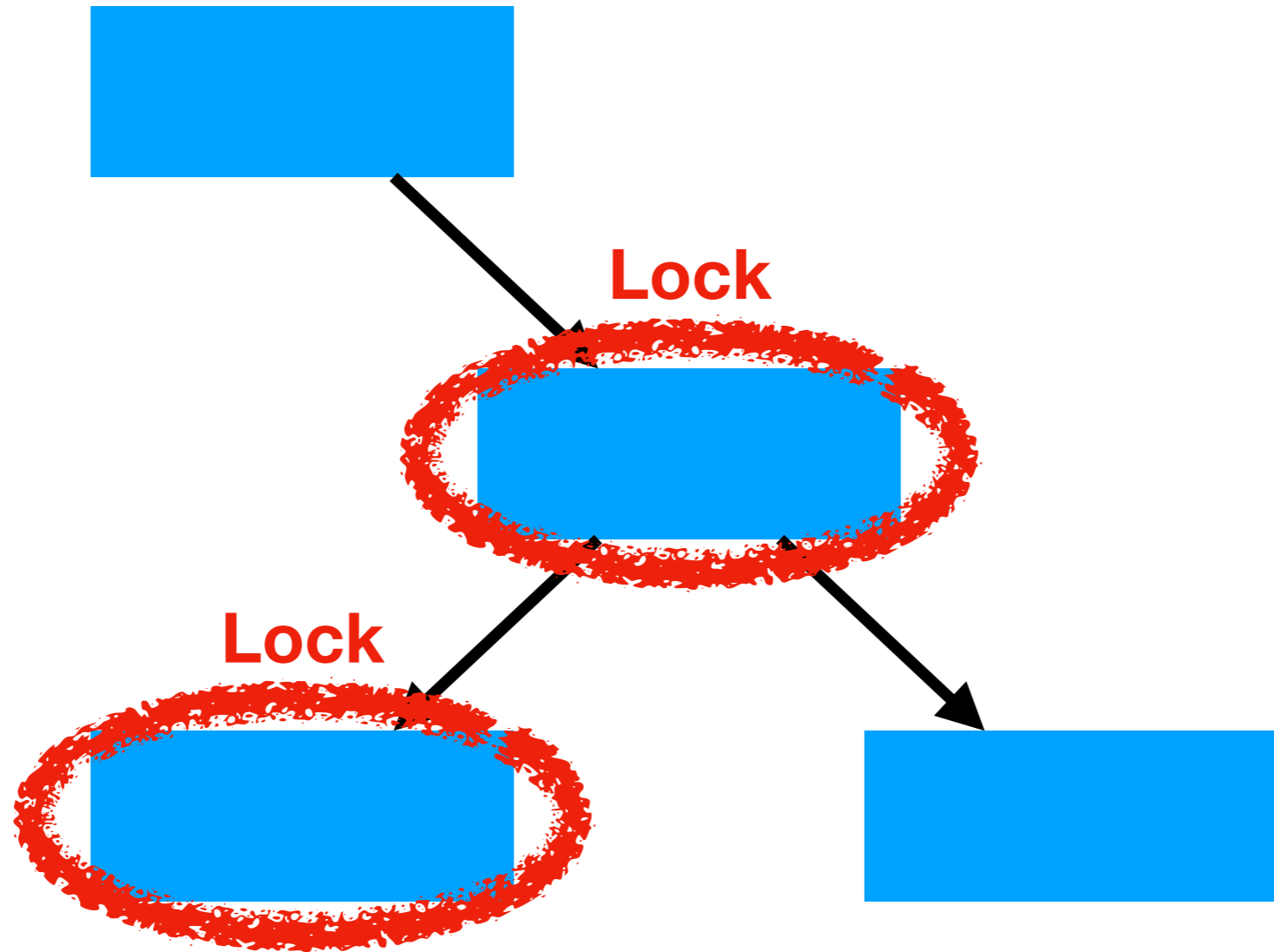
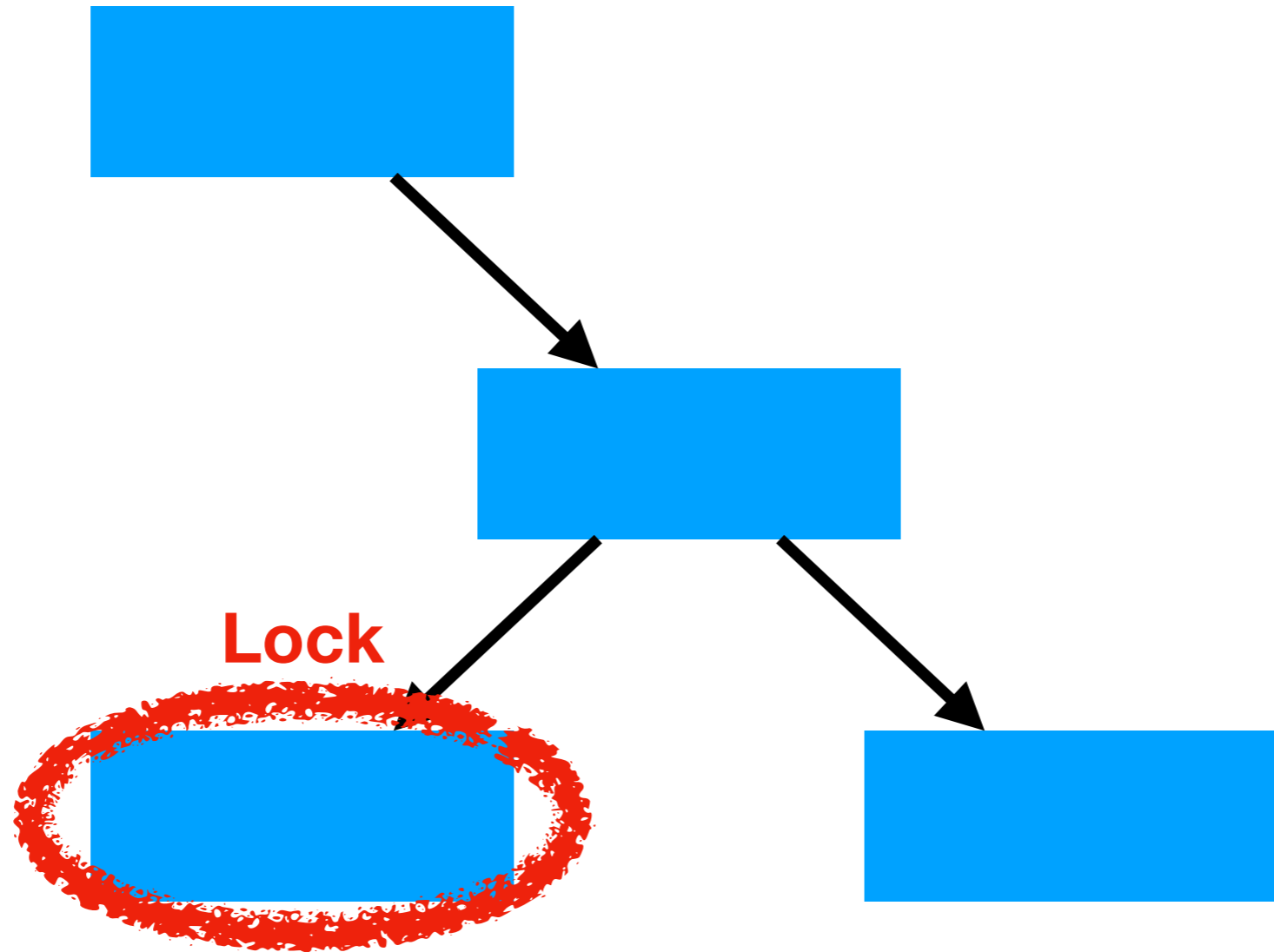


Illustration of Crabbing



Locking for Index Updates

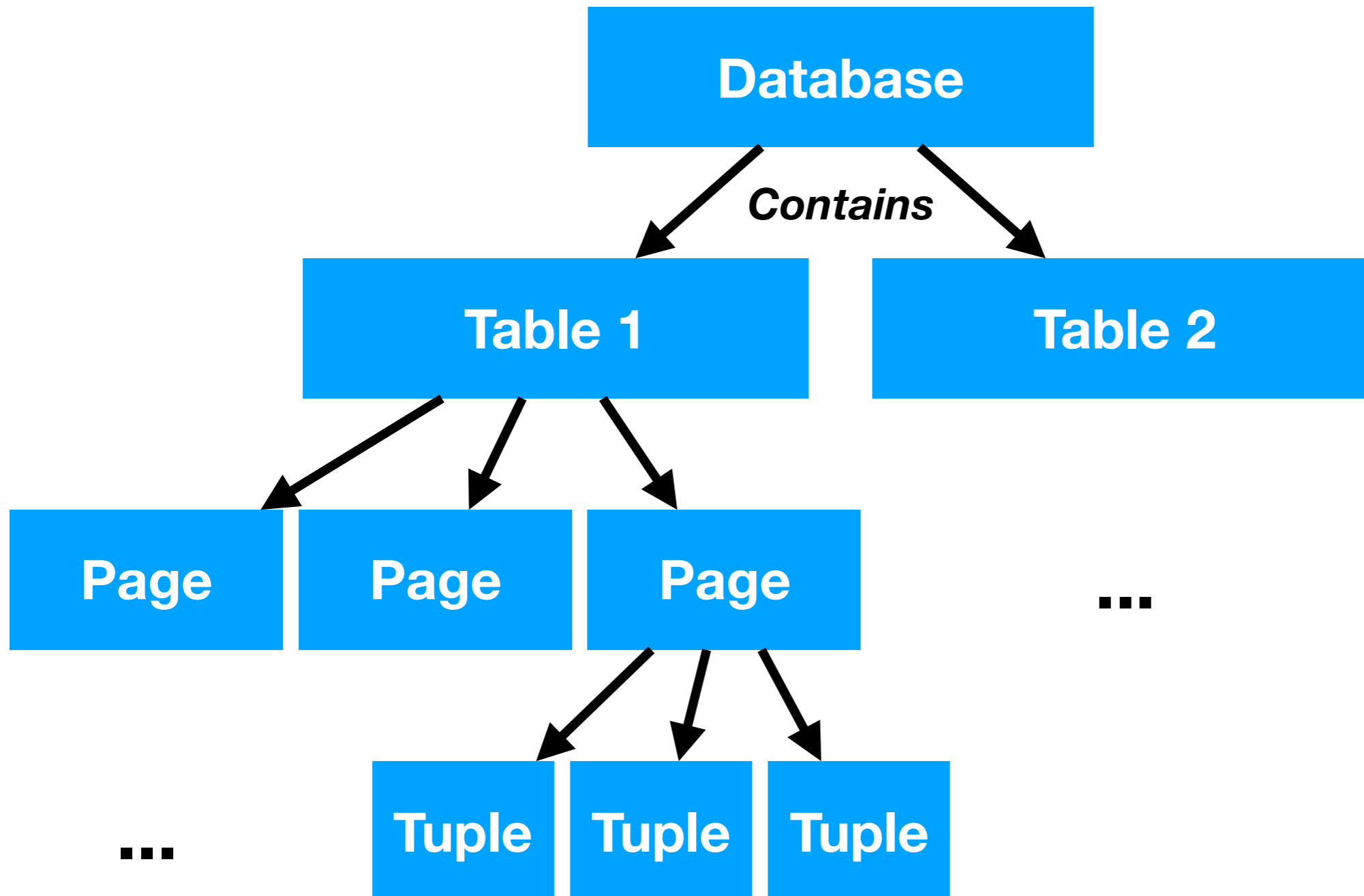
- So far: only considered index lookups; next: **updates**
- Index updates change index **leaf nodes**, may **propagate** up
- However, updates may not propagate upwards of "**safe**" nodes
 - Safe node is less than full (insertions)/more than half full (deletions)
- When traversing tree, **release prior locks** at each safe node
- May **pessimistically** request write locks but reduces performance
- Can **optimistically** request read locks for all nodes except leaf
 - Bets on **no propagation**, may have to restart if we lose

Multi-Granularity Locks

Multiple-Granularity Locks

- Fine-grained locking can **increase degree of parallelism**
- But fine-grained locking also **increases locking overheads**
- **Best granularity** may depend on query
 - E.g., whether we access **most** or **few** table rows
- Multiple-granularity locking **mixes lock granularities**
 - Have locks for entire table and locks for single rows
- Challenge: granting locks of diverse granularity **consistently**

Hierarchy of DB Objects



Multi-Granularity Locking

- **Cannot** treat locks at different granularities separately
 - May grant **conflicting** locks otherwise
- Need **locks on containing** objects before locking object
- Introduce new type of lock: **intention locks**
 - **IS (Intention Shared):**
want shared lock on contained object
 - **IX (Intention Exclusive):**
want exclusive lock on contained object

Lock Compatibility

	IS	IX	S	X
IS	✓	✓	✓	✗
IX	✓	✓	✗	✗
S	✓	✗	✓	✗
X	✗	✗	✗	✗

Lock Compatibility

	IS	IX	S	X
IS	✓	✓	✓	✗
IX	✓	✓	✗	✗
S	✓	✗	✓	✗
X	✗	✗	✗	✗

Lock Compatibility

Want shared lock on contained object

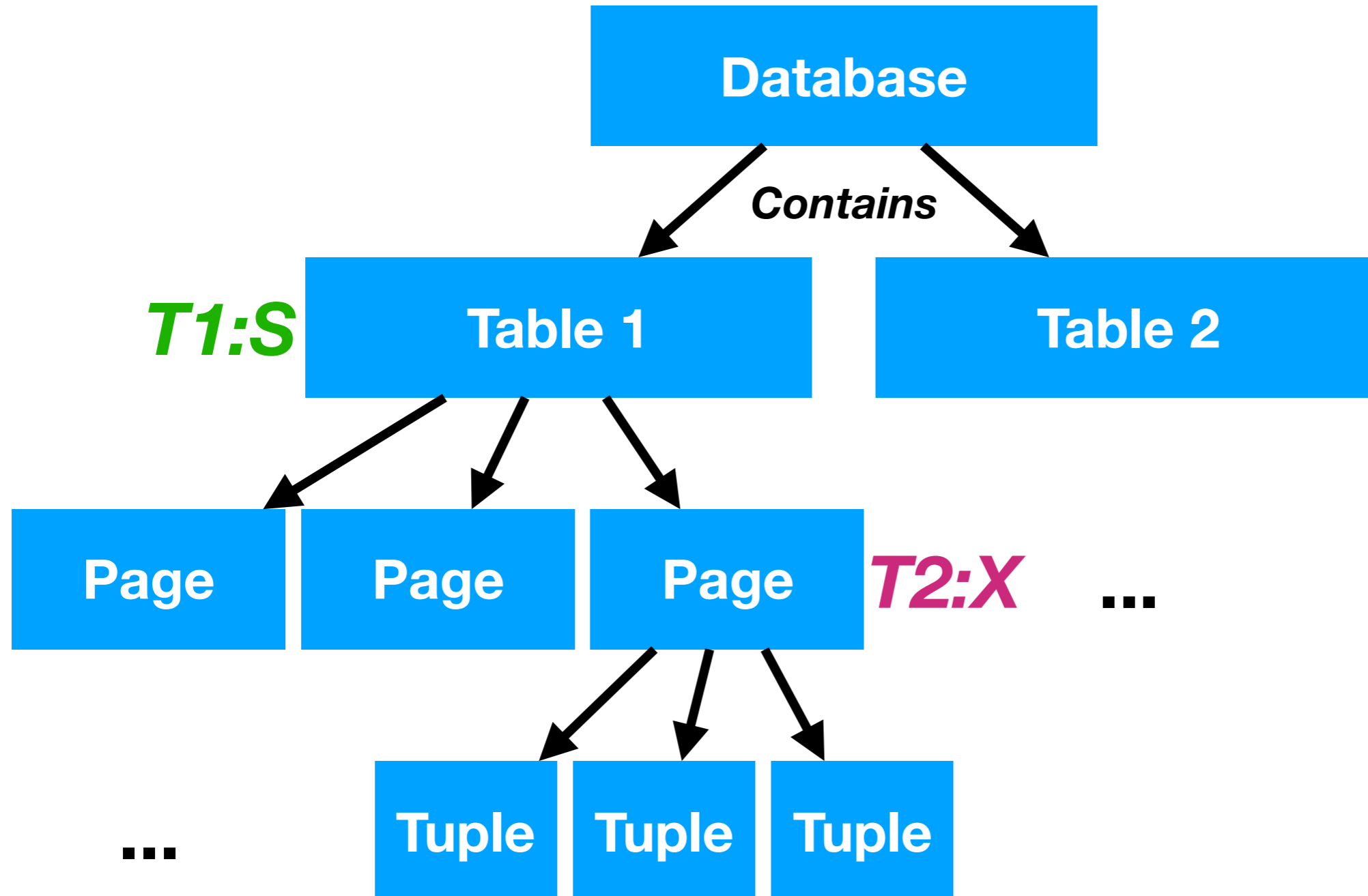
	IS	IX	S	X
IS	✓	✓	✓	✗
IX	✓	✓	✗	✗
S	✗	✗	✓	✗
X	✗	✗	✗	✗

Want exclusive lock on contained object

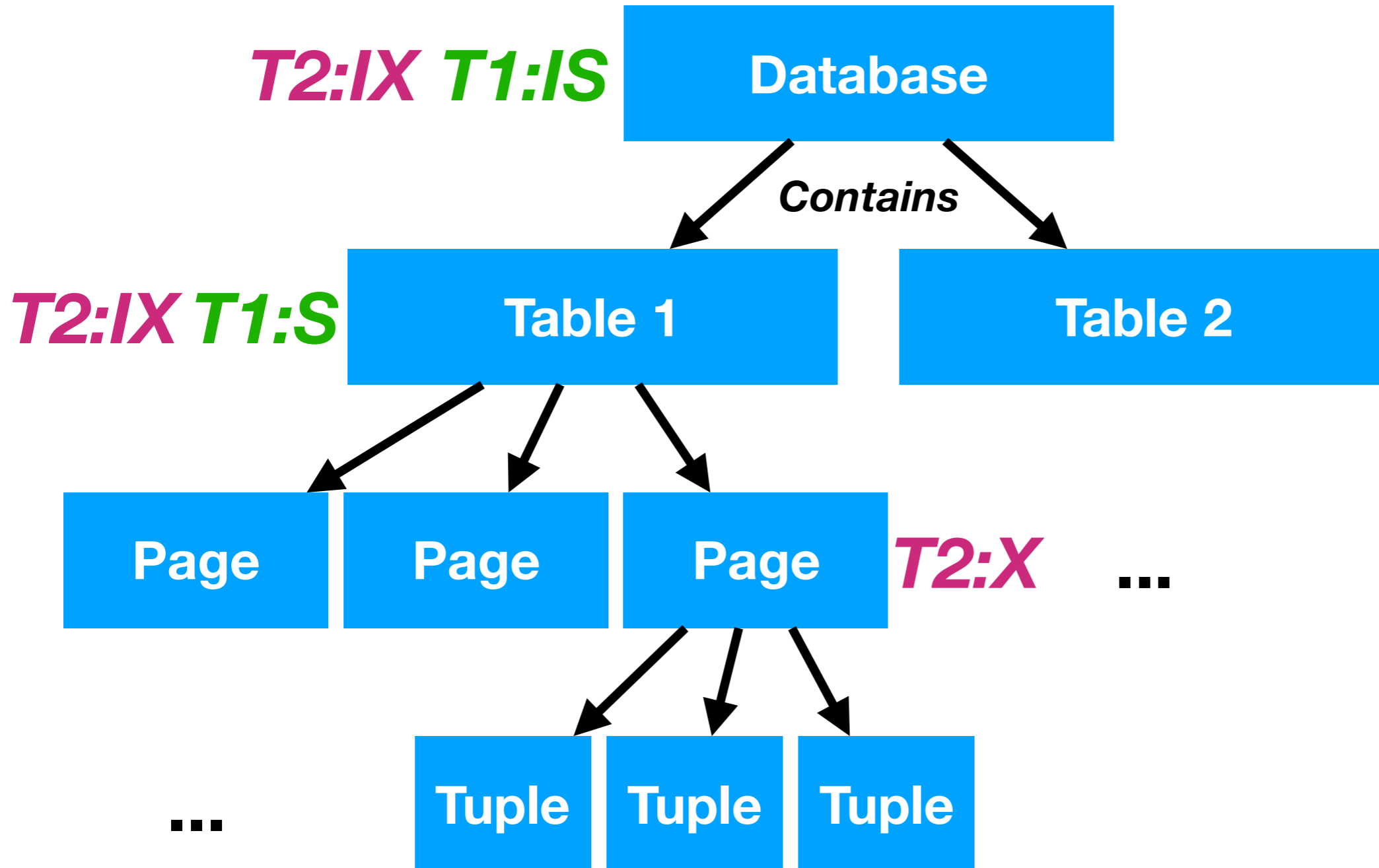
Using Intention Locks

- Need **IS** lock on **ancestors** before requesting Shared lock
- Need **IX** lock on **ancestors** before Exclusive lock
- Release intention locks from **leaf to root** node
 - Otherwise may have **inconsistent** locks

Inconsistent Locks



Intention Locks Help



Intention Locks Help

T2:IX T1:IS

Database

Contains

Table 1

Table 2

Page

Page

Page

T2:X

...

...

Tuple

Tuple

Tuple