Databases

Concurrency Control

Pierre Senellart

1 April 2026

Plan

Concurrent access to a database

- A DBMS serves many clients concurrently: Web applications, mobile apps, batch jobs, interactive users, etc.
- Each client performs a sequence of reads and writes
- Without coordination, concurrent accesses lead to inconsistent states and lost updates
- Example: two clients simultaneously book the last available room in a hotel $\rightarrow$ double booking
- The DBMS must provide isolation: each client's work appears to execute in isolation from all others
- This lecture: how does the DBMS achieve this? What guarantees are provided, and at what cost?

Transactions

- A transaction is a sequence of database operations treated as a single logical unit of work
- All-or-nothing: all operations succeed (commit), or all are undone (abort / rollback)

Transactions in SQL

```
BEGIN;                                  -- start a transaction
  UPDATE Account
    SET balance = balance - 100
    WHERE id = 'A';
  UPDATE Account
    SET balance = balance + 100
    WHERE id = 'B';
COMMIT;                                 -- make changes permanent
```

- Without explicit **BEGIN**, each statement is auto-committed (autocommit mode)
- **ROLLBACK** cancels all changes since **BEGIN**
- **SAVEPOINT** `sp;` / **ROLLBACK TO** `sp;` for partial rollback

ACID properties

Classical relational DBMS transactions satisfy ACID:

Atomicity. All operations in a transaction either all succeed, or are all rolled back. Mechanism: undo log, sometimes with MVCC.

Consistency. A transaction brings the database from one consistent state to another; integrity constraints are maintained. Mechanism: constraint checking.

Isolation. Concurrent transactions appear to execute serially. Mechanism: concurrency control (2PL, MVCC).

Durability. Once committed, changes are permanent, even after a crash. Mechanism: redo log (WAL).

Running example: bank transfers

- Two accounts tracked in a relation:

id	balance
A	1000
B	500

- T_1: transfer 100 € from A to B
- T_2: transfer 200 € from A to B
- Correct serial outcomes:
  - T_1 then T_2: $A = 700$, $B = 800$
  - T_2 then T_1: $A = 700$, $B = 800$ (same here)
- We will use this example to illustrate what can go wrong when T_1 and T_2 run concurrently

Plan

Why logging?

- A transaction may fail before commit: its partial effects must be undone (atomicity)
- The system itself may fail after commit: committed effects must not be lost (durability)
- The DBMS therefore keeps a log on stable storage, recording updates performed by transactions
- Typical log record for an update of object X:

$$\langle T_i, X, \text{old value}, \text{new value} \rangle$$

- This supports both:
  - undo: restore the old value
  - redo: reinstall the new value

Undo and redo

Undo. Used when a transaction is aborted or was active at crash time. Restore all modified objects to their old values.

Redo. Used after a crash for transactions that were already committed but whose updates may not yet have reached the data pages on disk. Reapply their new values.

Situation	Need undo?	Need redo?
Transaction aborted before commit	yes	no
Crash before commit	yes	no
Crash after commit, pages not flushed	no	yes
Crash after commit, pages flushed	no	no

Write-ahead logging (WAL)

- Principle: Before a modified data page is written to disk, the corresponding log record must already be on stable storage
- If the system crashes after the log is forced but before the data page is written:
  - redo can restore committed updates
  - undo can remove uncommitted ones
- Commit is acknowledged only after the transaction's commit record is forced to the log
- Data pages may be written later: improves performance
- Logging thus decouples correctness from the exact timing of page writes
- Systems differ in how undo is implemented: log-based undo or version-based undo (MVCC)
- ARIES [Mohan et al., 1992]: highly influential WAL algorithm

Example

Suppose the log contains:

$$\langle T_1, A, 1000, 900\rangle, \quad \langle T_1, \text{COMMIT}\rangle, \quad \langle T_2, B, 500, 700\rangle$$

- Crash occurs after these log records were forced, but before all modified data pages were written back
- T_1 was committed $\Rightarrow$ redo its update if needed
- T_2 was not committed $\Rightarrow$ undo its update
- Final recovered state reflects exactly the committed transactions

Plan

Schedules

- A schedule S for transactions $T_1, \ldots, T_n$ is an interleaving of their operations that preserves the relative order of operations within each transaction
- Operation notation:
  - $r_i(X)$: transaction T_i reads object X
  - $w_i(X)$: transaction T_i writes object X
  - c_i: T_i is committed; a_i: T_i is aborted
- A serial schedule: for every $i \neq j$, all operations of T_i complete before any operation of T_j starts, or vice-versa
- Serial schedules are always correct by definition
- Goal: identify interleaved schedules that are equivalent to some serial schedule, i.e., serializable schedules
- Full serializability may be too strong or too costly
- Hence weaker guarantees are often used: isolation levels (see later): Read Uncommitted, Read Committed, Repeatable Read, Snapshot Isolation, Serializable

Anomaly: dirty read

- T_2 reads a value written by T_1 before T_1 is committed; if T_1 is then aborted, T_2 has read data that never officially existed

Time	T_1	T_2
1	$r_1(A) \rightarrow 1000$	
2	$w_1(A \leftarrow 900)$	
3		$r_2(A) \rightarrow 900$ (dirty!)
4	ABORT	
5		$w_2(A \leftarrow 700), c_2$

- Result: $A = 700$ instead of 800; T_1's abort was ignored
- WR conflict (T_1 writes, T_2 reads, T_1 is aborted)
- Prevented by: Read Committed and above

Anomaly: lost update

- Two transactions read a value, compute an update, and write back; the second write overwrites the first

Time	T_1 (debit 100 €)	T_2 (debit 200 €)	
1	$r_1(A) \rightarrow 1000$		
2		$r_2(A) \rightarrow 1000$	
3	$w_1(A \leftarrow 900), c_1$		
4		$w_2(A \leftarrow 800), c_2$	(overwrites T_1!)

- Result: $A = 800$ instead of 700; T_1's update is lost
- WW conflict
- Prevented by: explicit locking or Repeatable Read and above

Anomaly: non-repeatable read

- T_1 reads the same object X twice and gets different values because T_2 modifies X in between

Time	T_1		T_2
1	$r_1(A) \rightarrow 1000$		
2			$r_2(A) \rightarrow 1000$
3			$w_2(A \leftarrow 800), c_2$
4	$r_1(A) \rightarrow 800$	(changed!)	
5	c_1		

- T_1 sees two different balances for A in the same transaction; decisions based on the first read may be wrong
- RW conflict
- Prevented by: Repeatable Read and above

Anomaly: phantom read

- T_1 executes a range query twice and gets different sets of rows because T_2 inserted/deleted rows matching the predicate in between

	T_1	T_2
1	`COUNT(*) WHERE room=101` $\rightarrow 0$	
2		`INSERT INTO Booking`, c_2
3	`COUNT(*) WHERE room=101` $\rightarrow 1$	(!)
4	c_1	

- Differs from non-repeatable read: not a modified row, but a new row appearing (a phantom)
- Not addressable by locking existing rows; requires predicate locks or next-key locks
- Prevented by: Serializable (and Repeatable Read in PostgreSQL's MVCC implementation)

Anomaly: write skew

- Each transaction reads a set of objects, checks an invariant, and writes to a disjoint set; the combined result violates the invariant
- Example: at least one of two doctors must be on call; both check and both go off call simultaneously

Time	T_1 (doctor Alice)	T_2 (doctor Bob)
1	$r_1(A) = \text{on}, r_1(B) = \text{on}$	
2		$r_2(A) = \text{on}, r_2(B) = \text{on}$
3	*check: other on-call, ok*	
4		*check: other on-call, ok*
5	$w_1(A \leftarrow \text{off}), c_1$	
6		$w_2(B \leftarrow \text{off}), c_2$

- Both committed; no one is on call: invariant violated
- Write sets are disjoint, different from lost update
- Not prevented by Snapshot Isolation! Requires Serializable

Plan

Conflicts

Definition

Two operations o_i and o_j from different transactions conflict if they access the same object X and at least one is a write:

- RW conflict: $r_i(X)$ followed by $w_j(X)$
- WR conflict: $w_i(X)$ followed by $r_j(X)$
- WW conflict: $w_i(X)$ followed by $w_j(X)$

Definition

Two schedules S and S' over the same transactions are conflict-equivalent if they contain the same operations and every pair of conflicting operations appears in the same relative order.

- Swapping non-conflicting adjacent operations preserves conflict-equivalence

Conflict-serializable schedules

Definition

A schedule S is conflict-serializable if it is conflict-equivalent to some serial schedule.

- Conflict-serializable $\Rightarrow$ same result as some serial execution $\Rightarrow$ correct
- Practical goal of concurrency control: produce only conflict-serializable schedules
- Question: how to efficiently test conflict-serializability?

Precedence graph

Definition

The precedence graph (or conflict graph) $G(S)$ of a schedule S:

- One node per transaction T_i
- A directed edge $T_i \to T_j$ $(i \neq j)$ if some operation of T_i conflicts with a later operation of T_j in S

Theorem (Serializability theorem)

A schedule S is conflict-serializable if and only if $G(S)$ is acyclic. A topological sort of $G(S)$ gives an equivalent serial order.

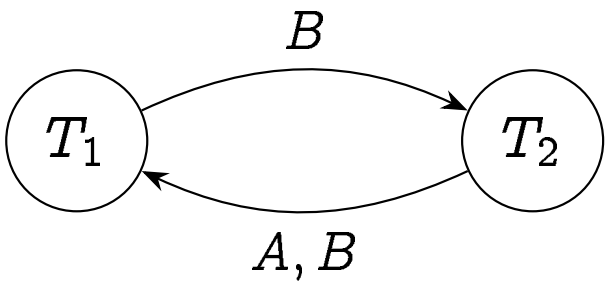
- Testing acyclicity: $O(n^2)$ in the number of transactions
- Efficient and practical

Precedence graph: example

Time	T_1	T_2
1	$r_1(A)$	
2		$r_2(A)$
3	$w_1(A)$	
4	$r_1(B)$	
5		$w_2(B)$
6	$w_1(B)$	
7	c_1	c_2

Conflicts:

- $r_2(A)$, $w_1(A)$: $T_2 \to T_1$
- $r_1(B)$, $w_2(B)$: $T_1 \to T_2$
- $w_2(B)$, $w_1(B)$: $T_2 \to T_1$



Cycle $\Rightarrow$ not conflict-serializable

View serializability

- A weaker notion than conflict-serializability
- Two schedules S, S' are view-equivalent if for every object X:
  1. Same initial reads: if T_i reads the initial value of X in S, it does so in S'
  2. Same dependent reads: if T_i reads X written by T_j in S, same in S'
  3. Same final writes: the same transaction performs the last write on X
- Conflict-serializable $\Rightarrow$ view-serializable (strictly, the converse fails)
- Testing view serializability is NP-complete [Papadimitriou, 1979]
- In practice: systems enforce conflict serializability (efficiently testable, slightly more restrictive)

Recoverability

- Conflict-serializability does not address aborts
- Recoverable schedule: if T_j reads a value written by T_i, then T_i is committed before T_j is committed – otherwise, committing T_j while T_i may still be aborted is dangerous
- Avoids Cascading Aborts (ACA): T_j only reads values written by already-committed transactions
- Strict schedule: additionally, T_j cannot write or read X until T_i (which last wrote X) has committed or aborted – allows recovery by restoring old values from the log
- Strict $\Rightarrow$ ACA $\Rightarrow$ recoverable
- Most practical systems enforce strict schedules

Plan

Locks

- A lock controls access to a database object (tuple, page, table, etc.)
- Two fundamental modes:

  Shared lock $S(X)$: for reading; multiple transactions can hold shared locks on X simultaneously

  Exclusive lock $X(X)$: for writing; incompatible with any other lock on the same object

	Shared	Exclusive
Shared	compatible	conflict
Exclusive	conflict	conflict

- A transaction blocks until it can acquire the requested lock
- Lock manager: central component maintaining a lock table

Two-phase locking (2PL)

- Naïve locking (acquire before use, release when done) does not guarantee conflict-serializability

Two-Phase Locking (2PL) Protocol

1. Growing phase: a transaction may acquire locks but not release any
2. Shrinking phase: a transaction may release locks but not acquire any

The point of maximum lock hold is the lock point

Theorem

Any schedule produced by transactions following 2PL is conflict-serializable. The serial order is given by the order of lock points.

2PL: proof sketch

- Proof by contradiction: suppose $G(S)$ has a cycle $T_1 \to T_2 \to \cdots \to T_k \to T_1$
- Edge $T_i \to T_{i+1}$: T_i holds a lock on X when T_{i+1} requests a conflicting lock on X
- T_{i+1} cannot acquire its lock until T_i enters its shrinking phase
- Therefore: lock point of T_i precedes lock point of T_{i+1}
- Following the cycle: lock point of T_1 precedes itself – contradiction □

- 2PL guarantees conflict-serializability but not recoverability: a transaction may release a lock and let another transaction read its uncommitted data

Strict 2PL

Strict 2PL

All exclusive locks are held until the transaction is committed or aborted

- Prevents dirty reads: no transaction can read data written by an uncommitted transaction
- Guarantees strict schedules: safe to abort by restoring old values (no cascading aborts)
- Strong Strict 2PL (Rigorous 2PL): hold *all* locks (shared and exclusive) until commit/abort; simpler to implement, less concurrency
- Most practical systems use Strict 2PL or Strong Strict 2PL
- Trade-off: holding locks longer reduces concurrency

Lock granularity and intention locks

- Locks can be acquired at multiple granularities: attribute $\subset$ tuple $\subset$ page $\subset$ table $\subset$ database
- Finer granularity $\rightarrow$ more concurrency, more overhead
- Problem: locking a table should prevent conflicting tuple-level locks without checking every tuple
- Intention locks resolve this:

  IS: intent to acquire shared lock on a descendant

  IX: intent to acquire exclusive lock on a descendant

  SIX: shared lock on this node + intent exclusive on descendants

- A transaction locks at coarse granularity with intention locks top-down, releases locks bottom-up
- Compatibility rules enforce that incompatible locks at the same level are detected without scanning descendants

Deadlocks

- A deadlock: a cycle of transactions each waiting for a lock held by the next
  - T_1 holds $X(A)$, requests $X(B)$
  - T_2 holds $X(B)$, requests $X(A)$
- Neither can proceed; neither will release voluntarily

Detection: Wait-For Graph

- Node per active transaction
- Edge $T_i \to T_j$ if T_i waits for T_j
- Periodically detect cycles
- Choose a victim and abort it; victim restarts with a new transaction

Prevention: Timestamps

- Assign timestamp $ts(T_i)$ at start
- Wait-Die: older waits for younger; younger is killed
- Wound-Wait: older preempts younger; younger waits for older
- No cycle can form; aborted T restarts with same ts

Plan

Timestamp ordering

- An alternative to locking: assign each transaction T_i a unique timestamp $ts(T_i)$ at start (e.g., system clock, logical counter)
- For each object X: maintain $W\text{-}ts(X)$ (largest ts that wrote X) and $R\text{-}ts(X)$ (largest ts that read X)

Basic Timestamp Ordering (TO) Protocol

- Read by T_i: if $ts(T_i) < W\text{-}ts(X)$, abort T_i (too late); else execute, update $R\text{-}ts(X)$
- Write by T_i: if $ts(T_i) < R\text{-}ts(X)$ or $ts(T_i) < W\text{-}ts(X)$, abort T_i; else execute, update $W\text{-}ts(X)$

- Guarantees conflict-serializability equivalent to the serial order by timestamp
- No deadlocks (aborted transactions restart with a new timestamp), but risk of starvation

Thomas's write rule

- When T_i writes X and $ts(T_i) < W\text{-}ts(X)$:
  - Basic TO: abort T_i
  - Thomas's rule: ignore (skip) the write; a later transaction already overwrote X, so T_i's write is obsolete
- Produces view-serializable (not always conflict-serializable) schedules; strictly more concurrent

Optimistic Concurrency Control (OCC)

- Read phase: execute freely on a private copy
- Validation phase: check for conflicts at commit time by comparing read/write sets
- Write phase: if valid, install changes
- Efficient when conflicts are rare; high abort rate under contention

Plan

Motivation: readers vs. writers

- Under locking (2PL): readers block writers (shared lock prevents exclusive lock) and vice versa
- Read-heavy workloads suffer: long-running reads block all writes on affected pages
- Key insight: instead of one version per object, keep multiple versions – one per modifying transaction
- Multi-Version Concurrency Control (MVCC):
  - Readers never block writers
  - Writers never block readers
  - Writers may still block concurrent writers
- Dominant approach in production systems: PostgreSQL, Oracle, MySQL InnoDB, SQL Server, CockroachDB, ...

MVCC mechanics (PostgreSQL)

- Each tuple row carries version metadata:
  - `xmin`: transaction ID that created this version
  - `xmax`: transaction ID that deleted or updated this version (∞ if current)
- **UPDATE**: insert a new tuple version + set `xmax` of the old version
- Transaction T (with ID xid) sees a version if:
  - $xmin \leqslant xid$ and the $xmin$ transaction has committed
  - $xmax > xid$ or the $xmax$ transaction has not committed
- Garbage collection: **VACUUM** removes tuple versions no longer visible to any active transaction
- Inspect visibility:
  SELECT `xmin, xmax, *` **FROM** `Account;`

Snapshot isolation

- Natural isolation level built on MVCC: Snapshot Isolation (SI)
- At transaction start, take a consistent snapshot of the database (set of all committed versions at that moment)
- All reads during the transaction use this fixed snapshot
- A transaction T can commit if no other committed transaction has written to any object that T has also written (first-committer-wins)

Properties of Snapshot Isolation

- Prevents: dirty reads, lost updates, non-repeatable reads, phantom reads
- Does not prevent: write skew
- Not equivalent to any SQL-92 level; sits strictly between Repeatable Read and Serializable [Berenson et al., 1995]

Write skew under snapshot isolation

- In SI: two transactions T_1 and T_2 take the same snapshot; both check the invariant on their snapshot; both pass the first-committer-wins check (write sets are disjoint)
- Cause: an rw-anti-dependency: T_j writes an object that T_i read from its snapshot (i.e., T_j "retroactively" invalidates T_i's read)
- Workarounds under SI:
  - **SELECT** ... **FOR UPDATE**: acquire an exclusive lock on the rows read $\rightarrow$ prevents concurrent conflicting writes
  - Materialize the conflict: introduce a surrogate row that both transactions write
  - Use Serializable snapshot isolation (SSI)

Serializable snapshot isolation (SSI)

- [Fekete et al., 2005, Cahill et al., 2008]: add anomaly detection on top of SI without the cost of full locking
- Key theorem [Fekete et al., 2005]: every non-serializable SI execution contains a cycle in the dependency graph involving two consecutive rw-anti-dependency edges
- SSI instruments transactions to detect such dangerous structures and aborts one transaction upon detection (conservative: may produce false positives)
- Result: serializable execution with throughput close to snapshot isolation for typical workloads
- PostgreSQL has implemented SSI since version 9.1 [Cahill et al., 2008]
- Application must retry serialization-failure errors

Plan

SQL standard isolation levels

SQL-92 defines four isolation levels by the anomalies they permit:

Level	Dirty read	Non-rep. read	Phantom read	Write skew
Read Uncommitted	yes	yes	yes	yes
Read Committed	no	yes	yes	yes
Repeatable Read	no	no	yes	yes
Serializable	no	no	no	no

- The standard only specifies which anomalies are permitted; implementations may prevent more
- [Berenson et al., 1995]: the SQL-92 characterizations are ambiguous and do not capture snapshot isolation – a widely used level that fits nowhere in this table

Setting the isolation level in SQL

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- or:
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Explicit row-level locking in SQL

```sql
-- Acquire exclusive lock on selected rows
-- (until COMMIT/ROLLBACK):
SELECT balance FROM Account WHERE id = 'A' FOR UPDATE;

-- Acquire shared lock on selected rows:
SELECT balance FROM Account WHERE id = 'A' FOR SHARE;

-- Table-level lock (for administrative operations):
LOCK TABLE Account IN EXCLUSIVE MODE;
```

PostgreSQL isolation implementation

PostgreSQL implements all levels via MVCC (+ SSI for Serializable):

SQL Level	Snapshot	Anomalies prevented
Read Uncommitted	per stmt	(same as Read Committed)
Read Committed	per stmt	dirty reads
Repeatable Read	per txn	+ non-rep. reads, phantoms
Serializable	per txn + SSI	all, incl. write skew

- PostgreSQL never uses Read Uncommitted behavior (treats it as Read Committed)
- Read Committed is the default; sufficient for many workloads
- Repeatable Read in PostgreSQL also prevents phantom reads (stronger than the SQL-92 standard requires)
- Serialization failures and deadlocks raise errors; application must retry

Constraints vs. snapshot visibility

- In Read Committed, each statement reads from a snapshot of committed data
- However, constraints (UNIQUE, foreign keys, CHECK) must hold for the current global state of the database

Key idea

Isolation controls what a transaction sees, but constraints restrict what it is allowed to write

- Example (UNIQUE constraint):
  - T_1: inserts $(x = 1)$, not yet committed
  - T_2: does not see this row in its snapshot
  - T_2: attempts to insert $(x = 1)$
- The DBMS detects the conflict using indexes and locks (not snapshots)
- T_2 will wait or abort, even though it could not see T_1's row

PostgreSQL: SSI in action

Concurrent booking anomaly detected and aborted:

Session 1

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT COUNT(*) FROM Reservation
  WHERE room = 101 AND arrival = '2026-04-08'; -- returns 0
INSERT INTO Reservation VALUES (42, 1, 101, '2026-04-08', 1);
COMMIT; -- succeeds
```

Session 2 (concurrent with Session 1)

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SELECT COUNT(*) FROM Reservation
  WHERE room = 101 AND arrival = '2026-04-08'; -- returns 0
INSERT INTO Reservation VALUES (43, 2, 101, '2026-04-08', 1);
COMMIT; -- Serialization ERROR
```

Choosing an isolation level

- Read Committed (default): read-mostly workloads; for read-modify-write, use `SELECT ... FOR UPDATE` to prevent lost updates on specific rows
- Repeatable Read: transaction needs a consistent snapshot (reports, long computations); in PostgreSQL also prevents phantom reads
- Serializable: complex invariants across multiple rows; application must retry on conflict error

Retry skeleton (Python/psycopg2)

```
for attempt in range(MAX_RETRIES):
    try:
        cur.execute(
            "BEGIN ISOLATION LEVEL SERIALIZABLE")
        do_work(cur)          # application logic
        cur.execute("COMMIT")
        break
    except SerializationFailure:
        cur.execute("ROLLBACK")
else:
    raise RuntimeError("too many retries")
```

Plan

References

- Classic reference on concurrency control and recovery theory: [Bernstein et al., 1987]
- Comprehensive modern treatment (theory and algorithms): [Weikum and Vossen, 2001]
- Concurrency Control chapter of the DB systems textbook [Garcia-Molina et al., 2008]
- Concurrency control in PostgreSQL: `https://www.postgresql.org/docs/current/mvcc.html`

Bibliography I

Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, pages 1–10, 1995.

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proc. SIGMOD*, pages 729–738, 2008.

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, 2008.

Bibliography II

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17 (1):94–162, 1992.

Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.