# DBMS

## UNIT - 5

**Database and Implementation**

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

# DATABASE SECURITY

## Threats to DB

1. Loss of integrity (unwanted modification)
2. Loss of availability (required access revoked)
3. Loss of confidentiality

## Security Mechanisms

- Principle of least privilege – multiuser DB system
- DBMS: database security and authorisation subsystem
- Two types of security mechanisms

### (a) Discretionary Access Control

- Grant privileges to users
- Access specific data (files, records, fields) in a specified mode (read, insert, delete, update)

### (b) Mandatory Access Control

- Enforce multilevel security
- Classify users & data into different security classes
- Eg: permit users at a certain level to access data from its level and all the levels below its level
- Eg: role-based security

## SQL: GRANT

```
GRANT privileges ON object TO users [WITH GRANT OPTION]
```

privileges that can be specified

- SELECT
- INSERT (col-name)
- INSERT
- DELETE
- REFERENCES (col-name)
- REFERENCES

## SQL : REVOKE

```
REVOKE [GRANT OPTION FOR] privileges ON object
FROM users {RESTRICT | CASCADE}
```

# TRANSACTION PROCESSING

- Transaction: executing program that forms a logical unit of DB processing
  - includes one or more DB access operations (IRUD)
  - specify boundaries with begin transaction and end transaction statements
  - can be read-only or read-write

- **Transaction processing systems:** systems with large DBs and concurrent users

## DATABASE MODEL

- **Database:** collection of named data items
- **Granularity:** size of data item
- **Data item:** DB record, disk block, individual field of a record

- Simplified model ; each item has unique ID

## Database Operations

(1) **read-item(x)** : reads DB item x into prog variable (also named x)

— steps:
→ find addr of disk block that contains item x
→ copy block into a buffer in main memory
→ copy item x from buffer to program variable x

(2) **write-item(X)** : writes prog variable x into DB item x

— steps:
→ find addr of disk block that contains item x
→ copy disk block into buffer in main memory
→ copy prog variable x into item x's location in the buffer
→ store updated disk block from buffer back to disk

- read-set of a transaction : set of items a transaction reads

- write-set of a transaction : set of items a transaction writes

## DBMS Buffers

- DBMS maintains several data buffers in main memory
- Altogether called database cache
- Each buffer: stores 1 disk block
- If buffers all occupied and new block read, buffer replacement policy used
  - Least Recently Used (LRU)

## CONCURRENCY CONTROL PROBLEMS

- Example DB: airline reservation DB
  - each record: one flight's no. of reserved seats

(a)

| $T_1$ |
|---|
| read_item($X$); |
| $X := X - N$; |
| write_item($X$); |
| read_item($Y$); |
| $Y := Y + N$; |
| write_item($Y$); |

(b)

| $T_2$ |
|---|
| read_item($X$); |
| $X := X + M$; |
| write_item($X$); |

**Figure 20.2**
Two sample transactions.
(a) Transaction $T_1$.
(b) Transaction $T_2$.

## (1) LOST UPDATE PROBLEM

- 2 transactions access same DB have operations interleaved such that values of some DB items are wrong

- Eg: read before write

(a)

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$; | |
| | read_item($X$);<br>$X := X + M$; |
| write_item($X$);<br>read_item($Y$); | |
| | write_item($X$); |
| $Y := Y + N$;<br>write_item($Y$); | |

Time ↓

Item $X$ has an incorrect value because its update by $T_1$ is *lost* (overwritten).

## (2) TEMPORARY UPDATE / DIRTY READ PROBLEM

- Transaction updates a DB item and fails
- Another transaction reads dirty item before it is rolled back

(b)

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($Y$); | |

Time ↓

Transaction $T_1$ fails and must change the value of $X$ back to its old value; meanwhile $T_2$ has read the *temporary* incorrect value of $X$.

## (3) INCORRECT SUMMARY PROBLEM

- Transaction 1 is calculating aggregate summary on DB items
- Transaction 2 is updating DB items

(c)

| $T_1$ | $T_3$ |
|---|---|
| | *sum* := 0;<br>read_item(*A*);<br>*sum* := *sum* + *A*;<br>⋮ |
| read_item(*X*);<br>*X* := *X* − *N*;<br>write_item(*X*); | |
| | read_item(*X*);<br>*sum* := *sum* + *X*;<br>read_item(*Y*);<br>*sum* := *sum* + *Y*; |
| read_item(*Y*);<br>*Y* := *Y* + *N*;<br>write_item(*Y*); | |

← $T_3$ reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

## (4) UNREPEATABLE READ PROBLEM

- Transaction 1 reads item twice
- Value changed in between reads by another transaction

| T1 | T2 |
|---|---|
| Read(X) | |
| | Read(X) ← |
| Write(X) | |
| | Read(X) ← |

different X values

## (5) PHANTOM READ PROBLEM

- Transaction reads an item twice
- Item deleted between reads
- Error thrown

| T1 | T2 |
| --- | --- |
| Read(X) | |
| | Read(X) |
| Delete(X) | |
| | Read(X) |

**Phantoms.** A transaction $T_1$ may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction $T_2$ inserts a new row $r$ that also satisfies the WHERE-clause condition used in $T_1$, into the table used by $T_1$. The record $r$ is called a **phantom record** because it was not there when $T_1$ starts but is there when $T_1$ ends. $T_1$ may or may not see the phantom, a row that previously did not exist. If the equivalent serial order is $T_1$ followed by $T_2$, then the record $r$ should not be seen; but if it is $T_2$ followed by $T_1$, then the phantom record should be in the result given to $T_1$. If the system cannot ensure the correct behavior, then it does not deal with the phantom record problem.

# Transaction & System Concepts

## (a) Transaction Operations

1. BEGIN_TRANSACTION
   - marks beginning

2. READ or WRITE
   - read or write operations on DB executed as a part of a transaction

3. END_TRANSACTION
   - specifies that read/write operations have ended
   - here, check whether changes in transaction to be permanently applied (committed) or aborted

## 4. COMMIT_TRANSACTION

- successful end of transaction
- all changes safely committed to DB

## 5. ROLLBACK (OR ABORT)

- ended unsuccessfully
- changes undone

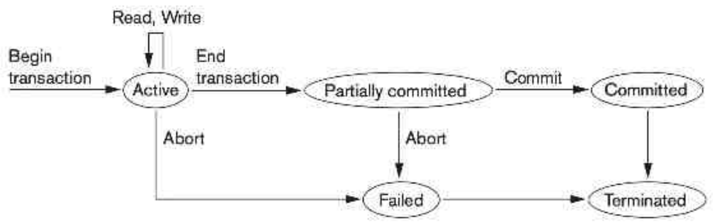



**Figure 20.4**
State transition diagram illustrating the states for transaction execution.

## (b) System Log

- Log keeping track of transaction operations
- Sequential, append-only file kept on disk
- Log buffers in memory with last part of the log file
  - when buffers filled, appended to file on disk
- Types of log records

1. **[start_transaction, *T*].** Indicates that transaction *T* has started execution.
2. **[write_item, *T*, *X*, *old_value*, *new_value*].** Indicates that transaction *T* has changed the value of database item *X* from *old_value* to *new_value*.
3. **[read_item, *T*, *X*].** Indicates that transaction *T* has read the value of database item *X*.
4. **[commit, *T*].** Indicates that transaction *T* has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. **[abort, *T*].** Indicates that transaction *T* has been aborted.

## Desirable Properties of Transactions

### ACID

- **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
- **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.
- **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
- **Durability or permanency.** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

## Characterising Schedules Based on Recoverability

- Schedule/ history: S of n transactions $T_1, T_2, \dots, T_n$ is ordering of operations of the transactions, interleaved
- Total ordering of transactions: order of operations in S said to be total ordering if for any two operations, one occurs before the other
- Shorthand

**(a)**

Time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$; | |
| | read_item($X$);<br>$X := X + M$; |
| write_item($X$);<br>read_item($Y$); | |
| | write_item($X$); |
| $Y := Y + N$;<br>write_item($Y$); | |

$S_a$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

**(b)**

Time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($Y$); | |

$S_b$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; $a_1$;

- **Conflicting Operations in a Schedule:** if they satisfy all 3
  1. belong to diff transactions
  2. access the same item X
  3. at least one of the operations is a write-item(X)

  - Eg: $S_a$ : $r_1(X)$ and $w_2(X)$ } read-write conflict
    $r_2(X)$ and $w_1(X)$
    $w_1(X)$ and $w_2(X)$ } write-write conflict

  - 2 operations conflict if changing their order can result in a different outcome

- **Complete Schedule:** if all 3 hold

1. The operations in $S$ are exactly those operations in $T_1, T_2, \ldots, T_n$, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction $T_i$, their relative order of appearance in $S$ is the same as their order of appearance in $T_i$.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.[10]

[10]Theoretically, it is not necessary to determine an order between pairs of *nonconflicting* operations.

- **Recoverable schedule:** once committed, a transaction never needs to be rolled back
  - S is recoverable if no T in S commits until all other transactions T' that have written some item X that T reads are committed

$S_a'$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $w_1(Y)$; $c_1$;

$S_a'$ is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory (see Section 20.5). However, consider the two (partial) schedules $S_c$ and $S_d$ that follow:

$S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $a_1$;
$S_d$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $c_1$; $c_2$;
$S_e$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $a_1$; $a_2$;

$S_c$ is not recoverable because $T_2$ reads item $X$ from $T_1$, but $T_2$ commits before $T_1$ commits. The problem occurs if $T_1$ aborts after the $c_2$ operation in $S_c$; then the value of $X$ that $T_2$ read is no longer valid and $T_2$ must be aborted *after* it is committed, leading to a schedule that is *not recoverable*. For the schedule to be recoverable, the $c_2$ operation in $S_c$ must be postponed until after $T_1$ commits, as shown in $S_d$. If $T_1$ aborts instead of committing, then $T_2$ should also abort as shown in $S_e$, because the value of $X$ it read is no longer valid. In $S_e$, aborting $T_2$ is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule $S_c$.

- **Cascadeless Schedule:** every transaction in S reads only items written by committed transactions
  - no cascading rollback will occur

- **Strict Schedule:** every transaction in S can neither read nor write an item X until the last transaction that wrote X has committed
  - recover: before image

# Characterising Schedules Based on Serialisability

## (1) Serial Schedule

- All operations of every T in S are executed consecutively in S
- No interleaving

(a) Time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$);<br>read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |

Schedule A

(b) Time ↓

| $T_1$ | $T_2$ |
|---|---|
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($X$);<br>$X := X - N$;<br>write_item($X$);<br>read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |

Schedule B

## (2) Serialisable Schedule

- Equivalent to serial schedule

### (a) Result equivalence

- produce the same result (state) of DB
- not always foolproof

**Figure 20.6**
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

| $S_1$ |
|---|
| read_item($X$);<br>$X := X + 10$;<br>write_item($X$); |

| $S_2$ |
|---|
| read_item($X$);<br>$X := X * 1.1$;<br>write_item ($X$); |

### (b) Conflict equivalence of two schedules

- relative order of conflicting operations is the same

(c) View equivalence
- more complex: later

- Serialisable schedule: S is conflict equivalent to a serial schedule S'

- Reorder non-conflicting operations on S until S' is formed

- A & D are equivalent (conflict)

(a)

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$);<br>read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |

Time ↓

Schedule A

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |

Time ↓

Schedule D

- ∴ D is serialisable

- A & C not equivalent

(a)

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$);<br>read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |

Time ↓

Schedule A

(c)

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$; | |
| | read_item($X$);<br>$X := X + M$; |
| write_item($X$);<br>read_item($Y$); | |
| | write_item($X$); |
| $Y := Y + N$;<br>write_item($Y$); | |

Time ↓

Schedule C

# Testing for Serialisability

- Construct a precedence/serialisation graph $G(N, E)$
- Each node is a transaction
- Each edge $e_i : T_j \rightarrow T_k$ for a pair of conflicting operations where it appears first in $T_j$ and then in $T_k$

**Algorithm 20.1.** Testing Conflict Serializability of a Schedule $S$

1. For each transaction $T_i$ participating in schedule $S$, create a node labeled $T_i$ in the precedence graph.
2. For each case in $S$ where $T_j$ executes a read_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a read_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in $S$ where $T_j$ executes a write_item($X$) after $T_i$ executes a write_item($X$), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule $S$ is serializable if and only if the precedence graph has no cycles.

## Q: Draw precedence graphs for the following

(a) Time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$);<br>read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |

Schedule A

(b) Time ↓

| $T_1$ | $T_2$ |
|---|---|
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($X$);<br>$X := X - N$;<br>write_item($X$);<br>read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |

Schedule B

(c)

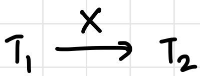
| Time ↓ $T_1$ | $T_2$ |
| --- | --- |
| read_item($X$);<br>$X := X - N$; | |
| | read_item($X$);<br>$X := X + M$; |
| write_item($X$);<br>read_item($Y$); | |
| | write_item($X$); |
| $Y := Y + N$;<br>write_item($Y$); | |

**Schedule C**

| Time ↓ $T_1$ | $T_2$ |
| --- | --- |
| read_item($X$);<br>$X := X - N$;<br>write_item($X$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); | |

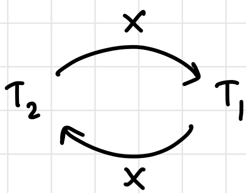
**Schedule D**

(A)



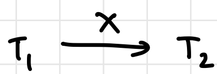

(B)


(C)




not serialisable

(D)

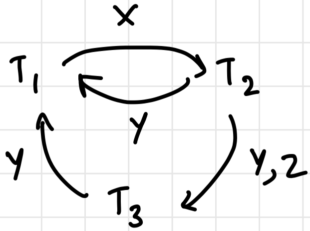



Q: Draw precedence graph

(b)

| Time ↓ Transaction $T_1$ | Transaction $T_2$ | Transaction $T_3$ |
| --- | --- | --- |
| | read_item($Z$);<br>read_item($Y$);<br>write_item($Y$); | |
| | | read_item($Y$);<br>read_item($Z$); |
| read_item($X$);<br>write_item($X$); | | |
| | | write_item($Y$);<br>write_item($Z$); |
| | read_item($X$); | |
| read_item($Y$);<br>write_item($Y$); | | |
| | write_item($X$); | |

**Schedule E**



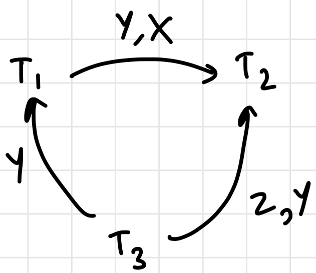

non-serialisable

(c)

Time ↓

| Transaction $T_1$ | Transaction $T_2$ | Transaction $T_3$ |
|---|---|---|
| | | read_item(Y);<br>read_item(Z); |
| read_item(X);<br>write_item(X); | | |
| | | write_item(Y);<br>write_item(Z); |
| | read_item(Z); | |
| read_item(Y);<br>write_item(Y); | | |
| | read_item(Y);<br>write_item(Y);<br>read_item(X);<br>write_item(X); | |

**Schedule F**




serialisable

## VIEW EQUIVALENCE

- All 3 conditions

1. The same set of transactions participates in $S$ and $S'$, and $S$ and $S'$ include the same operations of those transactions.
2. For any operation $r_i(X)$ of $T_i$ in $S$, if the value of $X$ read by the operation has been written by an operation $w_j(X)$ of $T_j$ (or if it is the original value of $X$ before the schedule started), the same condition must hold for the value of $X$ read by operation $r_i(X)$ of $T_i$ in $S'$.
3. If the operation $w_k(Y)$ of $T_k$ is the last operation to write item $Y$ in $S$, then $w_k(Y)$ of $T_k$ must also be the last operation to write item $Y$ in $S'$.

- Constrained write assumption:
  - any write operation $w_i(X)$ in $T_i$ is preceeded by a $r_i(X)$ in $T_i$ and the value written by $w_i(X)$ in $T_i$ depends only on the value of X read by $r_i(X)$
  - computation of new X is function f(X) on old X
  - opposite of blind write

## TRANSACTION SUPPORT IN SQL

- PSQL: begin; statements; commit/abort/rollback <sp>;
- Every transaction has either rollback, abort or commit
- Characteristics of transaction set by set transaction statement

(1) Access mode

- read only
- read write – default (except for read uncommitted)

### (2) Diagnostic area size

- no. of conditions (n) that can be simultaneously held in the diagnostic area
- supply feedback info on n most recent SQL statements

### (3) Isolation Level

- read uncommitted
- read committed
- repeatable read
- serialisable – no dirty read, unrepeatable read, phantom reads

**Table 20.1** Possible Violations Based on Isolation Levels as Defined in SQL

| | Type of Violation | | |
|---|---|---|---|
| **Isolation Level** | **Dirty Read** | **Nonrepeatable Read** | **Phantom** |
| READ UNCOMMITTED | Yes | Yes | Yes |
| READ COMMITTED | No | Yes | Yes |
| REPEATABLE READ | No | No | Yes |
| SERIALIZABLE | No | No | No |

# CONCURRENCY CONTROL

- Enforce isolation
- DB consistency

## 1. Two-Phase Locking Techniques

- Lock(X): requesting transaction locks item X
- Unlock(X): item X made available to all transactions

## — 1.1 Binary Locking

- Two states: locked and unlocked
- Distinct lock with each DB item X
- If value of lock on DB item X = 1, X cannot be accessed by operations requesting it
- If value = 0, can be accessed and locked
- Current value of X's lock: lock(X)

- Operations: lock-item(X) and unlock-item(X) — atomic

```
lock_item(X):
B:  if LOCK(X) = 0                 (*item is unlocked*)
        then LOCK(X) ←1            (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
```

```
unlock_item(X):
    LOCK(X) ← 0;                   (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

- Lock table: record of items currently locked

- Lock manager: manages locks, stores into lock table

## — 1.2 Shared / Exclusive (Read/Write) Locks

- Several read accesses or single write access
- Multiple-mode lock
- Operations: read-lock(X), write-lock(X), unlock(X)
- lock(X) has 3 possible states

- Lock table entries:

lock(X) value

<data-item_name, lock, no-of-reads, locking-transactions>

- Atomic operations

- read-lock(X)

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
```

- write-lock(X)

```
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
```

- unlock(X)

```
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

- Lock Conversion

  - Upgrading: if T is the only transaction holding a read lock on X at the time it issues a request for a write lock, the lock can be upgraded

  - Downgrading: if T holds a write lock on X at the time when it issues a request for a read lock, it can be downgraded

  - Operation definitions to be modified to account for this functionality

- Locking alone does not guarantee serialisability

(a)

| $T_1$ | $T_2$ |
|---|---|
| read_lock($Y$); | read_lock($X$); |
| read_item($Y$); | read_item($X$); |
| unlock($Y$); | unlock($X$); |
| write_lock($X$); | write_lock($Y$); |
| read_item($X$); | read_item($Y$); |
| $X := X + Y$; | $Y := X + Y$; |
| write_item($X$); | write_item($Y$); |
| unlock($X$); | unlock($Y$); |

(b) Initial values: $X$=20, $Y$=30

Result serial schedule $T_1$ followed by $T_2$: $X$=50, $Y$=80

Result of serial schedule $T_2$ followed by $T_1$: $X$=70, $Y$=50

(c)

Time ↓

| $T_1$ | $T_2$ |
|---|---|
| read_lock($Y$);<br>read_item($Y$);<br>unlock($Y$); | |
| | read_lock($X$);<br>read_item($X$);<br>unlock($X$);<br>write_lock($Y$);<br>read_item($Y$);<br>$Y := X + Y$;<br>write_item($Y$);<br>unlock($Y$); |
| write_lock($X$);<br>read_item($X$);<br>$X := X + Y$;<br>write_item($X$);<br>unlock($X$); | |

Result of schedule $S$:
$X$=50, $Y$=50
(nonserializable)

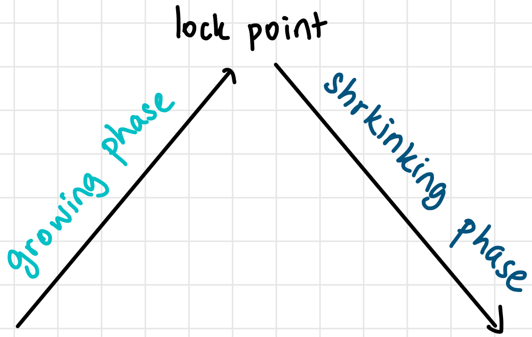
**Figure 21.3**
Transactions that do not obey two-phase locking. (a) Two transactions $T_1$ and $T_2$. (b) Results of possible serial schedules of $T_1$ and $T_2$. (c) A nonserializable schedule $S$ that uses locks.

## TWO-PHASE LOCKING PROTOCOL

- All locking operations precede the first unlock operation in the transaction
- Two phases: expanding/growing phase, where locks can only be acquired, and shrinking phase, where only existing locks can be released and no locks can be acquired
  - upgrading of locks in expanding phase
  - downgrading of locks in shrinking phase

**Figure 21.4**
Transactions $T_1'$ and $T_2'$, which are the same as $T_1$ and $T_2$ in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

| $T_1'$ | $T_2'$ |
|---|---|
| read_lock($Y$); | read_lock($X$); |
| read_item($Y$); | read_item($X$); |
| write_lock($X$); | write_lock($Y$); |
| unlock($Y$) | unlock($X$) |
| read_item($X$); | read_item($Y$); |
| $X := X + Y$; | $Y := X + Y$; |
| write_item($X$); | write_item($Y$); |
| unlock($X$); | unlock($Y$); |




## 2. Variations of 2PL Systems

### 2.1 Basic (described above)

### 2.2 Conservative / static: T must lock all items it accesses before trancaction begins execution (to prevent deadlocks)

- Predeclare read_set and write_set of items
- If cannot lock any one item, does not lock any items

### 2.3 Strict: T does not release any write locks until after it commits or aborts

- strict schedule for recoverability
- not deadlock-free

2.4 Rigorous : T does not release any locks until after it commits or aborts
- easier to implement than strict
- expanding phase until it ends

# Deadlock Prevention

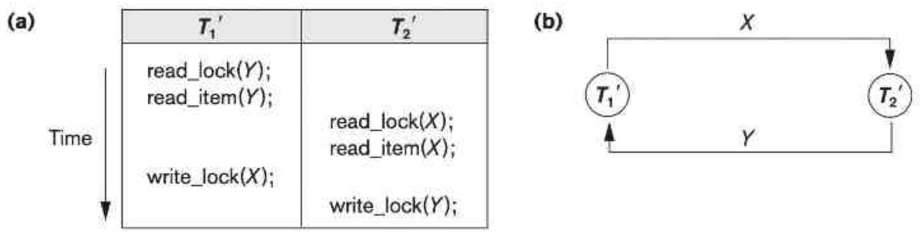



**Figure 21.5**
Illustrating the deadlock problem. (a) A partial schedule of $T_1'$ and $T_2'$ that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

- Transaction timestamp TS(T') : smaller for older transactions, unique for every transaction

## Protocols

1. Lock in advance
   - If any one not available, lock none

2. Ordered locking
   - Lock DB items in a specific order

3. Wait-die
- $T_i$ tries to lock item X but cannot because $T_j$ is holding it
- If $TS(T_i) < TS(T_j)$, $T_i$ is allowed to wait
- Otherwise, abort $T_i$ (younger) and restart later with the same timestamp
- New old allowed to wait, new young killed

4. Wound-wait
- $T_i$ tries to lock item X but cannot because $T_j$ is holding it
- If $TS(T_i) < TS(T_j)$, abort $T_j$ (younger) and restart with same timestamp
- Otherwise, $T_i$ (younger) allowed to wait
- New young allowed to wait, new old wounds existing young

5. No waiting

6. Cautious waiting

## DEADLOCK DETECTION

1. Wait-for graph
- If cycles present, deadlock

2. Timeouts
- If $T_i$ waits for longer than threshold, abort transaction and assume it was deadlocked

# Concurrency Control Based on Timestamp Ordering

- Serialisable in same order as order of timestamps
- Timestamp ordering (TO)
- Each item X has 2 timestamp values: read-TS(X) and write-TS(X)

1. **read_TS(X).** The **read timestamp** of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_TS(X) = TS(T), where T is the *youngest* transaction that has read X successfully.
2. **write_TS(X).** The **write timestamp** of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, write_TS(X) = TS(T), where T is the *youngest* transaction that has written X successfully. Based on the algorithm, T will also be the last transaction to write item X, as we shall see.

## 1. Basic TO

- If transaction T tries to issue a read-item(X) or a write-item(X), the value of TS(T) is compared with read-TS(X) and write-TS(X)

- If ordering violated, abort and rollback (cascading rollback)

1. Whenever a transaction T issues a write_item(X) operation, the following check is performed:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).
2. Whenever a transaction T issues a read_item(X) operation, the following check is performed:
   a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already written the value of item X before T had a chance to read X.
   b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the *larger* of TS(T) and the current read_TS(X).

- No deadlocks
- Starvation

## 2. Strict TO

- Transaction T issues a read-item(X) or write-item(X) where TS(T) > write-TS(X) (younger)
- read/write operation delayed until T' commits or aborts (TS(T') = write-TS(X))
- Simulate locking
- No deadlocks; T waits for T' only if TS(T) > TS(T')
- Same with write-item(X) and read-TS(X)
- Conflict serialisability and strict

## 3. Thomas' Write Rule

- Does not enforce conflict serialisability

1. If read_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation.
2. If write_TS(*X*) > TS(*T*), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(*T*)—and hence after *T* in the timestamp ordering—has already written the value of *X*. Thus, we must ignore the write_item(*X*) operation of *T* because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(*X*) operation of *T* and set write_TS(*X*) to TS(*T*).

# NOSQL

- Many reads, min writes
- Semi-structured

## Characteristics related to distributed DBs and systems

1. Scalability (horizontal scaling)
2. Availability
3. Replication models (master-slave, master-master)
4. Sharding of files (horizontal partitioning)
5. High performance data access (hashing or range partitioning)

## Characteristics related to data models and query languages

1. Not requiring schema
2. Less powerful query languages
3. Versioning

## Types of NOSQL Systems

1. Document-based (MongoDB, Couch DB, Raven, Terra Store)
2. Key-value stores (Berkeley DB, LevelDB, Memcached, Redis, Riak)
3. Column-based (Cassandra, Amazon SimpleDB, Hbase, Hypertable)
4. Graph-based (FlockDB, Neo4j, Orient, Infinite Graph)

# All in the NoSQL Family

NoSQL databases are geared toward managing large sets of varied and frequently updated data, often in distributed systems or the cloud. They avoid the rigid schemas associated with relational databases. But the architectures themselves vary and are separated into four primary classifications, although types are blending over time.

### Document databases

Store data elements in document-like structures that encode information in formats such as JSON.

+

Common uses include content management and monitoring web and mobile applications.

+

EXAMPLES
Couchbase Server, CouchDB, MarkLogic, MongoDB

### Graph databases

Emphasize connections between data elements, storing related "nodes" in graphs to accelerate querying.

+

Common uses include recommendation engines and geospatial applications.

+

EXAMPLES
AllegroGraph, IBM Graph, Neo4j

### Key-value databases

Use a simple data model that pairs a unique key and its associated value in storing data elements.

+

Common uses include storing clickstream data and application logs.

+

EXAMPLES
Aerospike, DynamoDB, Redis, Riak

### Wide-column stores

Also called table-style databases, they store data across tables that can have very large numbers of columns.

+

Common uses include internet search and other large-scale web applications.

+

EXAMPLES
Accumulo, Amazon SimpleDB, Cassandra, HBase, Hypertable

## Comparision of Different NoSQL DBs

| Data model | Performance | Scalability | Flexibility | Complexity | Functionality |
|---|---|---|---|---|---|
| Key–value store | high | high | high | none | variable (none) |
| Column-oriented store | high | high | moderate | low | minimal |
| Document-oriented store | high | variable (high) | high | low | variable (low) |
| Graph database | variable | variable | high | high | graph theory |
| Relational database | variable | variable | low | moderate | relational algebra |

Q: What DB to use?

(a) Calculate average income relational
(b) Build shopping cart key-value
(c) Storing structured product information document
(d) Describing how user got from point A to B graph

# MONGODB

- Document (JSON)

```
> db.createCollection("courses")
{ "ok" : 1 }
> db.courses.find()
> db.courses.insert({"code": "UE19CS301", "name": "DBMS", "credits": 4})
WriteResult({ "nInserted" : 1 })
> db.courses.find()
{ "_id" : ObjectId("61b5ebb36aff316669c1258d"), "code" : "UE19CS301", "name" : "DBMS", "credits" : 4 }
```

↳ unique _id : auto indexed

- ObjectId: 12 bytes (4 + 3 + 2 + 3)

## Data Models

### 1. Embedded Data Model

- All related data in single document

(a) project document with an array of embedded workers:

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire",
    Workers: [
                { Ename: "John Smith",
                  Hours: 32.5
                },
                { Ename: "Joyce English",
                  Hours: 20.0
                }
             ]
};
```

### 2. Normalised data model

- 1NF

(c) normalized project and worker documents (not a fully normalized design for M:N relationships):

```
{
    _id:            "P1",
    Pname:          "ProductX",
    Plocation:      "Bellaire"
}
{   _id:            "W1",
    Ename:          "John Smith",
    ProjectId:      "P1",
    Hours:          32.5
}
```

# Commands

## 1. Use Command

- connect (or create and connect) to a DB

```
> use testdb
switched to db testdb
```

## 2. db command

- show connected database

```
> db
testdb
```

## 3. show dbs / databases

```
> show databases
admin          0.000GB
config         0.000GB
local          0.000GB
```

```
> show dbs
admin          0.000GB
config         0.000GB
local          0.000GB
```

## 4. db.dropDatabase()

```
> db.dropDatabase()
{ "ok" : 1 }
```

- Recall: lab week 1

**Q: Display first document in collection employee**

db.employee.findOne()

Q: Display the document of employee with empid=2

```
db.employee.findOne({"empid": 2})
```

Q: Return documents where birth is between 1940-01-01 and 1960-01-01

```
db.employees.find({"Birth": { $gt: new
Date('1940-01-01'), $lt: new Date('1960-01-01')}})
```

## KEY-VALUE DB

- No query language ; set of operations

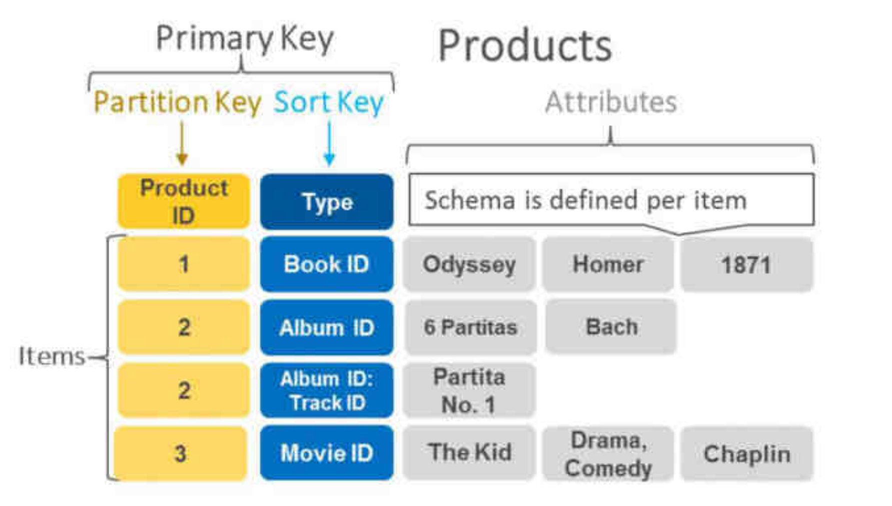

- Use cases
  - Storing Session Information
  - User Profiles, preferences
  - Shopping Cart Data
  - Article/Blog Comments
  - Product Categories/Details/Reviews
  - Internet Protocol Forwarding tables
  - Telecom directories

- **Key:** unique ID
- **Value:** text, number etc

## DYNAMODB

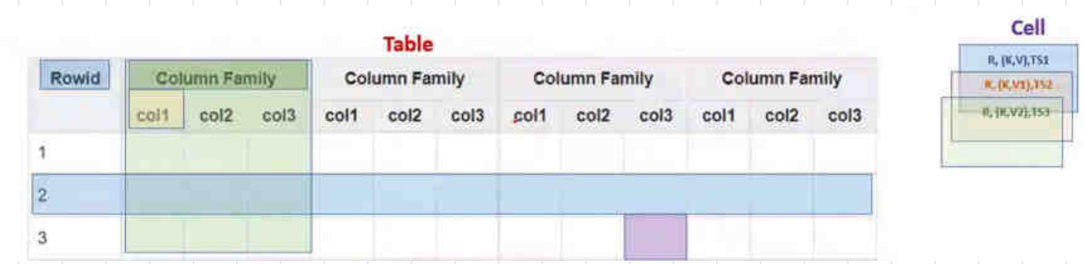
- Cloud-based, AWS
- Tables, items, attributes
- Number of attr-value pairs in an item
- Table name & primary key
- Types of PKs

- **A single attribute.** The DynamoDB system will use this attribute to build a hash index on the items in the table. This is called a *hash type primary key*. The items are not ordered in storage on the value of the hash attribute.
- **A pair of attributes.** This is called a *hash and range type primary key*. The primary key will be a pair of attributes (A, B): attribute A will be used for hashing, and because there will be multiple items with the same value of A, the B values will be used for ordering the records with the same A value. A table with this type of key can have additional secondary indexes defined on its attributes. For example, if we want to store multiple versions of some type of items in a table, we could use ItemID as hash and Date or Timestamp (when the version was created) as range in a hash and range type primary key.




## COLUMN-BASED

- Group columns into column families




- **Table**: Data represented as a collection of rows sorted on RowID
- **Row:** Collection of column families identified by **RowID** (Row Key), a byte array, serving as the primary key for the table and is indexed for fast lookup
- **Column:** Collection of key-value pairs – represented by ColumnFamilyName:ColumnName
- **Column family**: Collection of variable number columns
- **Cell**: Stores data and is a combination of {row key, column, timestamp/version} tuple as a byte array
- Timestamp (System timestamp) or any other unique version number within a RowId, for the cell

| Name | Data |
|---|---|
| HarryPotter | Info:{height:"4.5ft", age: "11@2011"}<br>School:{House:"Gryffindor", Sports:"Quidditch"} |
| Voldemort | Info:{height:"7ft", age: "50"}<br>School:{House:"Syltherin", Role:"Prefect"} |

## Hbase Usage

### 1. Create table

table name

```
hbase(main):001:0> create 'test', 'data'
0 row(s) in 0.9810 seconds
```

column family name

### 2. Insert values

table name

```
hbase(main):003:0> put 'test', 'row1', 'data:1', 'value1'
hbase(main):004:0> put 'test', 'row2', 'data:2', 'value2'
hbase(main):005:0> put 'test', 'row3', 'data:3', 'value3'
```

row key — column name — value

### 3. Retrieve values

specific row

```
hbase(main):006:0> get 'test', 'row1'
COLUMN                       CELL
 data:1                      timestamp=1414927084811, value=value1
1 row(s) in 0.0240 seconds
hbase(main):007:0> scan 'test'
ROW                          COLUMN+CELL
 row1                        column=data:1, timestamp=1414927084811, value=value1
```

all rows

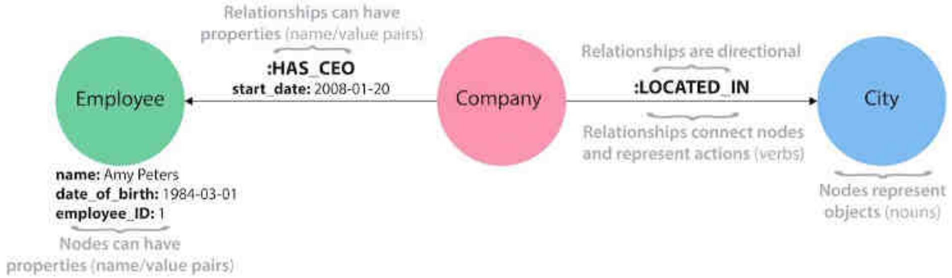
## 4. CRUD Operations

**(c) Some Hbase basic CRUD operations:**

Creating a table: create <tablename>, <column family>, <column family>, ...
Inserting Data: put <tablename>, <rowid>, <column family>:<column qualifier>, <value>
Reading Data (all data in a table): scan <tablename>
Retrieve Data (one item): get <tablename>,<rowid>

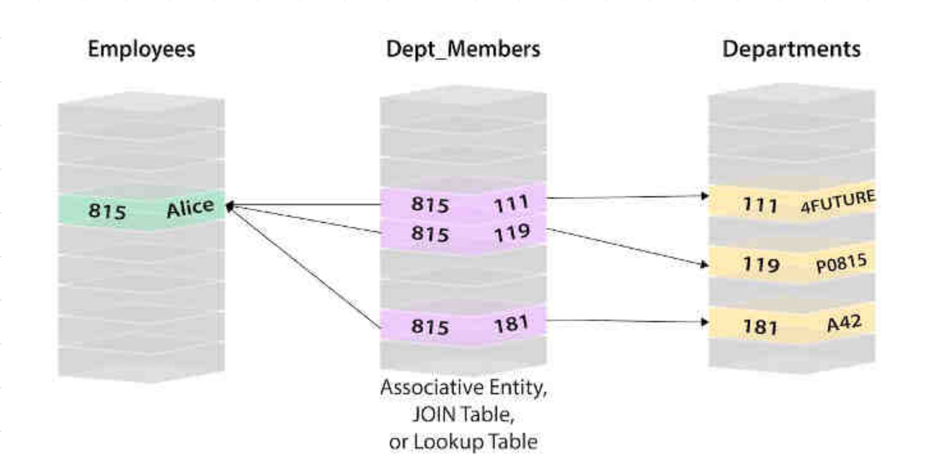
# GRAPH DB

- Relationships b/w data important
- RDBMS: join for relationships
  Graph DB: connections alongside data
- Neo4j: nodes and relationships
- Nodes: k-v pairs (any number) called properties
- Relationships: connections between node entities
  - direction
  - start node
  - end node
  - properties
- Cypher Query Language (CQL)

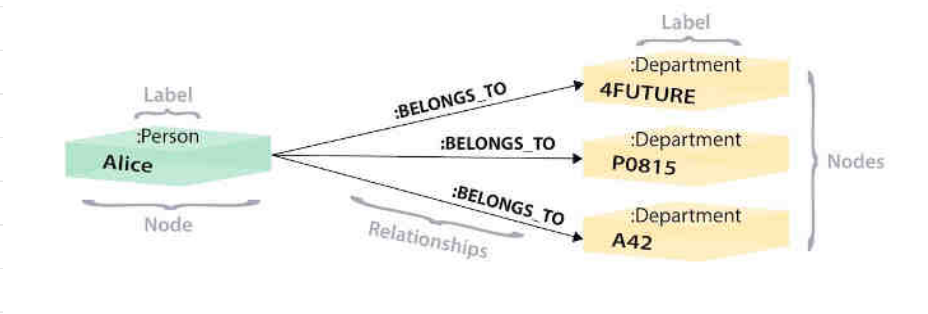


# Graph vs RDB

- RDBMS




- Graph




## Use Cases

- Fraud detection
- Network monitoring
- Recommendation engines

## Syntax

- Week 3 lab

**Figure 24.4**
Examples in Neo4j using the Cypher language. (a) Creating some nodes. (b) Creating some relationships.

**(a) creating some nodes for the COMPANY data (from Figure 5.6):**

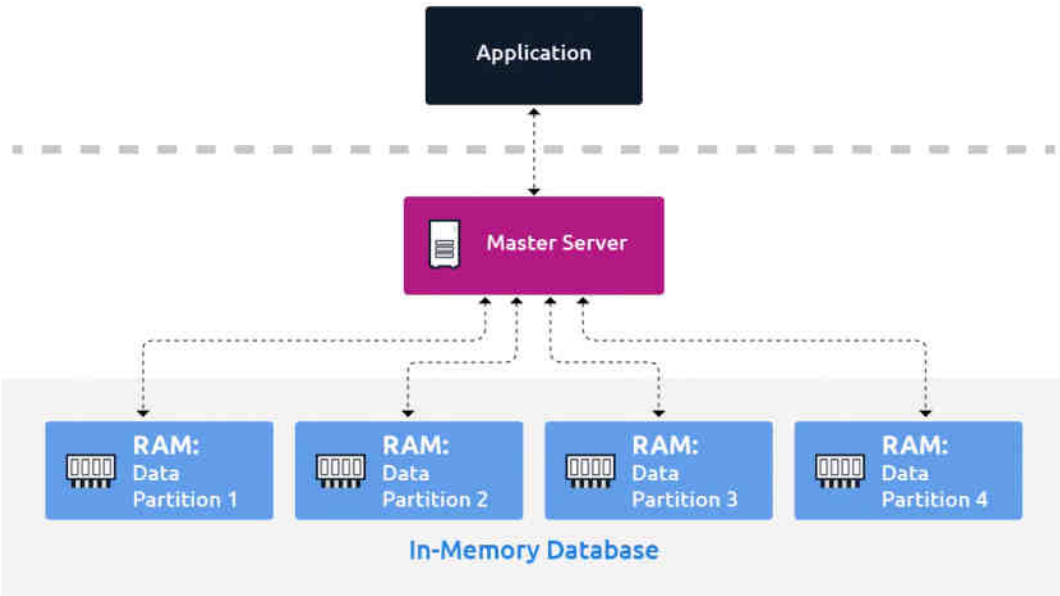
```
CREATE (e1: EMPLOYEE, {Empid: '1', Lname: 'Smith', Fname: 'John', Minit: 'B'})
CREATE (e2: EMPLOYEE, {Empid: '2', Lname: 'Wong', Fname: 'Franklin'})
CREATE (e3: EMPLOYEE, {Empid: '3', Lname: 'Zelaya', Fname: 'Alicia'})
CREATE (e4: EMPLOYEE, {Empid: '4', Lname: 'Wallace', Fname: 'Jennifer', Minit: 'S'})
...
CREATE (d1: DEPARTMENT, {Dno: '5', Dname: 'Research'})
CREATE (d2: DEPARTMENT, {Dno: '4', Dname: 'Administration'})
...
CREATE (p1: PROJECT, {Pno: '1', Pname: 'ProductX'})
CREATE (p2: PROJECT, {Pno: '2', Pname: 'ProductY'})
CREATE (p3: PROJECT, {Pno: '10', Pname: 'Computerization'})
CREATE (p4: PROJECT, {Pno: '20', Pname: 'Reorganization'})
...
CREATE (loc1: LOCATION, {Lname: 'Houston'})
CREATE (loc2: LOCATION, {Lname: 'Stafford'})
CREATE (loc3: LOCATION, {Lname: 'Bellaire'})
CREATE (loc4: LOCATION, {Lname: 'Sugarland'})
...
```

**(b) creating some relationships for the COMPANY data (from Figure 5.6):**

```
CREATE (e1) – [ : WorksFor ] –> (d1)
CREATE (e3) – [ : WorksFor ] –> (d2)
...
CREATE (d1) – [ : Manager ] –> (e2)
CREATE (d2) – [ : Manager ] –> (e4)
...
CREATE (d1) – [ : LocatedIn ] –> (loc1)
CREATE (d1) – [ : LocatedIn ] –> (loc3)
CREATE (d1) – [ : LocatedIn ] –> (loc4)
CREATE (d2) – [ : LocatedIn ] –> (loc2)
...
CREATE (e1) – [ : WorksOn, {Hours: '32.5'} ] –> (p1)
CREATE (e1) – [ : WorksOn, {Hours: '7.5'} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p1)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p2)
CREATE (e2) – [ : WorksOn, {Hours: '10.0'} ] –> (p3)
CREATE (e2) – [ : WorksOn, {Hours: 10.0} ] –> (p4)
...
```

## IN-MEMORY DATABASE

- In-memory storage and computation
- Fast
- Risk of data loss due to server failure




- Volt DB