Dos **operaciones son conflictivas** si operan sobre el mismo ítem y al menos una de ellas es una escritura.
Dos historias **Hi y Hj son conflicto equivalentes** Hi ≡ Hj sii 1. Están definidas sobre el mismo conjunto de transacciones 2. El orden de las operaciones conflictivas de transacciones no abortadas es el mismo.
Una historia H es **conflicto serializable (SR)** si es conflicto equivalente a alguna historia serial Hs
Testeo de Serializabilidad: Grafo de precedencia SG(H). Es un grafo dirigido con las siguientes características: 1. Un nodo para cada transacción Ti ⊆ H 2.Hay ejes entre Ti y Tj sii hay una operación de Ti que precede en H a una operación de Tj y son operaciones conflictivas. 3. Etiquetamos los ejes del grafo con los nombres de los ítems que los generan.
Teorema de la seriabilidad: Una historia H es SR sí y solo sí SG(H) es acíclico.
Si el SG(H) es acíclico entonces los órdenes seriales equivalentes son los diferentes órdenes topológicos del grafo.
Recuperación: Se puede asumir que el Abort se implementa recuperando imágenes anteriores de los ítems.
Dadas dos transacciones Ti y Tj decimos que **Ti lee X de Tj** si Ti lee X y Tj fue la última transacción que escribió X y no abortó antes de que Ti lo leyera. Es decir: 1. wj(X) < ri(X) 2. !(aj < ri(X)) 3 Si hay algún wk(X) tal que wj(X) < wk(X) < ri(X) entonces ak < ri(X)
Una historia H es **RC (recuperable)** si siempre que Ti lee de Tj con i ≠ j en H y ci ∈ H entonces cj < ci. Intuitivamente una historia es recuperable si una transacción realiza commit sólo después de que hicieron commit todas las transacciones de las cuales lee.
Una historia H es **ACA (avoid cascading aborts)** si siempre que Ti lee X de Tj con i ≠ j en H entonces cj < ri(X). Lee sólo valores de transacciones que ya hicieron commit.
Una historia H es **ST (estricta)** si siempre que wj(X) < oi(X) con i ≠ j entonces aj < oi(X) o cj < oi(A) siendo oi(X) igual a ri(X) o a wi(X) Es decir no se puede leer ni escribir un ítem hasta que la transacción que lo escribió previamente haya hecho commit o abort.
Teorema de la recuperabilidad: ST ⊂ ACA ⊂ RC
Ortogonalidad SR intersecta a todos los conjuntos RC, ACA y ST. Son conceptos ortogonales. Es fácil ver que una historia serial es también ST.
1. li(A) Lock. La transacción i realiza un bloqueo o lock sobre el ítem A. 2. ui(A) UnLock. La transacción i libera los bloqueos o locks previos sobre el ítem A. Usado en todos los modelos, se asume que libera todos los locks tomados.
Consistencia de Transacciones: 1. Una Ti puede leer o escribir un ítem X si previamente realizó un lock sobre X y no lo ha liberado 2. Si una transacción Ti realiza un lock sobre un elemento debe posteriormente liberarlo.
Legalidad: Una Ti que desea obtener un lock sobre X que ha sido lockeado por Tj en un modo que conflictua, debe esperar hasta que Tj haga unlock de X.
Grafo de precedencia para lock binario: Se asume H legal. Para hacer el SG(H) se siguen los siguientes pasos: 1 Hacer un nodo por cada Ti ⊆ H 2 Si Ti realiza un li(X) para algún ítem X y luego Tj con i ≠ j realiza un li(X) hacer un arco Ti → Tj
Lock ternario:
Consistencia (a) Una accion ri(X) debe ser precedida por un rli(X) o un wli(X), sin que intervenga un ui(X) (b) Un accion wi(X) debe ser precedida por una wli(X) sin que intervenga un ui(X) (c) Todos los locks deben ser seguidos de un unlock del mismo elemento

Lock Sostenido por T_j

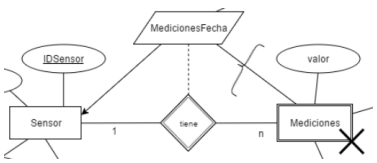
Lock solicitado por T_i		rl	wl
	rl	Si	No
	wl	No	No

Legalidad de las Historias (a) Si wli(X) aparece en una historia, entonces no puede haber luego un wlj(X) o rlj(X) para j ≠ i sin que haya primero un ui(X) (b) Si rli(X) aparece en una historia no puede haber luego un wlj(X) para j ≠ i sin que haya primero un ui(X)
Grafo de precedencia para Locking ternario 1. Hacer un nodo por cada Ti 2. Si Ti hace un rli(X) o wli(X) y luego Tj con j ≠ i hace un wlj(X) en H hacer un arco Ti → Tj 3 Si Ti hace un wli(X) y Tj con j ≠ i hace un rlj(X) en H entonces hacer un arco Ti → Tj. Básicamente dice que si dos transacciones realizan un lock sobre el mismo ítem y al menos uno de ellas es un write lock se debe dibujar un eje desde la primera a la segunda.
upgrading lock, es decir pasar de un lock de escritura o compartido a un lock exclusivo o de escritura.
Deadlock al usar upgrade lock Supongamos T1 y T2, y se presenta la siguiente historia donde cada una quiere realizar un upgrade lock. Ambos son denegados: H = rl1(X); rl2(X); wl1(X); wl2(X)
Update Lock Se puede evitar este deadlock si agregamos otro modo de lock llamado update lock. Un update lock sobre un ítem X que denotamos uli(X) da a la transacción Ti privilegio de lectura sobre X pero no de escritura. Como ventaja el update lock pasa a ser el único que puede ser upgraded a write lock
Una transacción respeta el **protocolo de bloqueo en dos fases (2PL)** si todas las operaciones de bloqueo (lock) preceden a la primer operación de desbloqueo (unlock) en la transacción. Una transacción que cumple con el protocolo se dice que es una transacción 2PL 1. Fase de crecimiento: toma los locks 2. Fase de contracción: libera los locks

		Lock Sostenido		
		rl	**wl**	**ul**
Lock Solicitado	**rl**	Si	No	No
	wl	No	No	No
	ul	Si	No	No

Serializabilidad con 2PL Dado T = T1, T2, ..., Tn, si toda Ti en T es 2PL, entonces todo H legal sobre T es SR.
2PL Estricto (2PLE o S2PL) Una transacción cumple con 2PL Estricto si es 2PL y no libera ninguno de sus locks de escritura hasta después de realizar el commit o el abort. **Serializabilidad con 2PL Estricto** 2PLE garantiza que la historia es ST. No es libre de deadlock.
2PL Riguroso (2PLR) Una transacción cumple con 2PL Riguroso si es 2PL y no libera ninguno de sus locks de escritura o lectura hasta después de realizar el commit o el abort. La **serializabilidad es igual al orden de los commit**
Detección de deadlock usando Wait-for Graph 1. Un nodo por cada transacción que tiene un lock o espera por uno. 2. Un eje entre dos nodos (Ti y Tj) si Ti está esperando que Tj libere un lock sobre un ítem que Ti necesita bloquear**.**
Si Ti intenta realizar un lock sobre un ítem y no puede porque Tj ya tiene un lock previo entonces hay dos estrategias**: Wait-Die** 1. Si TS(Ti) < TS(Tj) (Ti más viejo que Tj), entonces Ti se lo pone en espera 2. Si TS(Ti) > TS(Tj)(Ti más joven que Tj),entonces se aborta Ti (Ti dies) y se recomienza mas tarde con el mismo timestamp.
Wound-Wait 1. Si TS(Ti) < TS(Tj) (Ti más viejo que Tj), entonces, abortar Tj (Ti wounds Tj) y recomienza más tarde con el mismo timestamp. 2. Si TS(Ti) > TS(Tj)(Ti más joven que Tj),entonces, Ti se pone en espera. **Cautious waiting** (**CW**) Cuando Ti quiere bloquear un ítem que está bloqueado por Tj: Si Tj no está bloqueada (no esta esperando por algún otro ítem bloqueado) entonces Ti es bloqueado y espera. En otro caso Ti aborta
Timestamping:
Los timestamps determinan el orden de serialización. Cada elemento de la base de datos, X, debe asociarse a dos timestamp y un bit extra.1. RT(X): tiempo de lectura, el timestamp más alto de una transacción que ha leído X 2. WT(X): tiempo de escritura, el timestamp más alto de una transacción ha escrito X 3. C(X): bit de commit para X, es verdadero sii la transacción más reciente que escribió X ha realizado commit
Cada transacción se hubiera realizado **instantáneamente al momento del timestamp**. Si eso no ocurre entonces el comportamiento se denomina **físicamente irrealizable.**
Read too Late 1. TS(T) < WT(X) 2. Una transacción T intenta leer X pero el valor de escritura indica que X fue escrito después de que teóricamente debería haberlo leído T. **T aborta**
Write too Late 1. WT(X) < TS(T) < RT(X). 2. T intenta escribir pero el tiempo de lectura de X indica que alguna otra transacción debería haber leído el valor escrito por T (lee otro valor en su lugar). **T aborta**
Una **lectura sucia** ocurre cuando se le permite a una transacción la lectura de un elemento que ha sido modificado por otra transacción concurrente pero que todavía no ha sido cometida (commit). es mejor retrasar la lectura hasta que realice el commit o aborte
Thomas write rule La escritura puede “saltearse”cuando ya existe una escritura de una transacción con un timestamp de mayor valor. Es decir cuando WT(X) > TS(T). Cuando una transacción U escribe un elemento X, la escritura es tentativa y puede ser deshecha si U aborta. C(X) se pone falso y el planificador hace una copia de los valores de X y de WT(X) previos.
Ante la **solicitud de una transacción T** para una lectura o escritura, el planificador puede: 1Conceder la solicitud 2Abortar y reiniciar T con un nuevo timestamp (rollback) 3Demorar T y decidir luego si abortar o conceder lasolicitud (si el requerimiento es una lectura que podría ser sucia).
El **planificador recibe una solicitud de lectura rt (X):** Caso 1: Si TS(T) >= WT(X) - es físicamente realizable: 1Si C(X) es True, conceder la solicitud. Si TS(T)>RT(X) hacer RT(X)=TS(T), de otro modo no cambiar RT(X). 2Si C(X) es False demorar T hasta que C(X) sea verdadero o la transacción que escribió a X aborte.
El **planificador recibe una solicitud de escritura wt (X):**
Caso 1: Si TS(T) >= RT(X) y TS(T) >= WT(X) - es físicamente realizable 1. Escribir el nuevo valor para X 2. WT(X) := TS(T), o sea asignar nuevo WT a X. 3. C(X):= false, o sea poner en falso el bit de commit. **Caso 2:** Si TS(T) >= RT (X) pero TS(T) < WT (X) - es físicamente realizable, pero ya hay un valor posterior en X. 1Si C(X) es true, ignora la escritura. 2Si C(X) es falso demorar T hasta que C(X) sea verdadero o la transacción que escribió a X aborte **Caso 3**: Si TS(T) < RT(X) - es físicamente irrealizable, es decir, write too late. Se hace Rollback T (abortar y reiniciar con un nuevo timestamp).
El planificador recibe una **solicitud de commit C(T). 1.** Para cada uno de los elementos X escritos por T se hace: C(X) := true. **2.** Se permite proseguir a las transacciones que esperan a que X sea committed
El planificador recibe una **solicitud de abort o rollback A(T) o R(T).**Cada transacción que estaba esperando por un elemento X que T escribió debe repetir el intento de lectura o escritura y verificar si ahora el intento es legal
Generalmente el **timestamping es mejor** cuando la mayoría de las transacciones son de lectura o es raro que haya transacciones que traten de leer y escribir el mismo elemento. En situaciones de mucho conflicto, locking suele comportarse mejor.
Multiversion:
Evita aborts ocasionados por eventos read-too-late. si tj contiene la operación wj (x), podemos denotar la versión de x creada por esta escritura como xj Un **planificador multiversión** es un **planificador monoversión** si su función de versión asigna cada lectura a la última escritura precedente en el mismo elemento de datos.
Multiversión view serializable Sea m una historia multiversión, se dice que m es multiversión view serializable si existe una historia m′ serial monoversión tal que m ≈v m′ . MVSR es la clase de historias view serializables (serializables por vista) multiversión. Decidir si una historia esta en MSVR es un problema NP-Completo.
El **grafo de conflictos de una historia m** denotado como **G (m)** se construye con nodos por cada transacción de m con un eje ti → tj si rj (xi) esta en m. Para cualquier par de schedulers multiversión, m ≈v m′ entonces G (m) = G (m)′ .
Multiversión conflicto serializable (MCSR) Un conflicto multiversión en un schedule multiversión m es un par de pasos ri(xj) y wk(xk) tales que ri (xj) <m wk(xk)

El único **tipo relevante de conflictos son los pares de operaciones read-write** en el mismo ítem de datos, no necesariamente en la misma versión
Una **historia multiversión es multiversión reducible** si puede transformarse en una historia serial monoversión mediante una secuencia de pasos de transformación. Un historia multiversión m es multiversión conflicto serializable si hay una historia monoversión serial para el mismo conjunto de transacciones en el que todos los pares de operaciones en conflicto multiversión ocurren en el mismo orden que en m. Una historia multiversión es multiversión reducible sii es multiversión conflicto serializable
El **grafo de conflicto multiversión** de m es un grafo que tiene las transacciones de m como sus nodos y una arista de ti a tk si existen pasos ri (xj) y wk (xk) para el mismo elemento de datos x en m tal que ri (xj) <m wk (xk). Una historia multiversión es MCSR si y solo si su grafo de conflictos de multiversión es acíclico. La historia monoversión serial equivalente para un grafo de conflictos multiversión acíclico no se puede derivar simplemente ordenando topológicamente el grafo
Multiversion Timestamp Ordering (MVTO) Cada versión de un elemento de datos lleva un timestamp ts(ti) de la transacción ti que fue la que creo la versión. 1 Una operación ri (x) se transforma en una operación multiversión ri (wk) donde wk es la versión de x que tiene el timestamp mas grande menor o igual que ts(ti) y que fue escrita por tk , k ≠ i2. Una operación wi (x) se procesa de la siguiente manera: Si una operación rj (xk) tal que ts(tk) < ts(ti) < ts(tj) ya existe en el schedule entonces wi (x) es rechada y ti es abortada. Se produce un write too late.en otro caso wi (x) se transforma en wi (xi) y es ejecutada. 3. Un commit ci se retrasa hasta que los commit cj de todas las transacciones tj que han escrito nuevas versiones de los elementos de datos leídos por ti hayan sido ejecutados.
El **MV2PL** es un protocolo basado el locking usando strong strict two-phase locking o 2PL riguroso. Versiones de datos: commiteadas, actual (ultima version comiteada), versiones no comiteadas
Si el **paso no es el final** dentro de una transacción:1. Una lectura r (x) se ejecuta de inmediato, asignándole la versión actual del
elemento de datos solicitado o asignándole una versión no commiteada de x 2. Un escritura w (x) se ejecuta solamente cuando la transacción que ha escrito x por última vez finalizo, es decir no hay otra versión no commiteada de x. O sea se libero el lock de escritura sobre x Si es el **paso final** de la transacción ti , esta se retrasa hasta que commitean las siguientes transacciones 1. todas aquellas transacciones tj que hayan leído la versión actual de un elemento de datos escrito por ti 2. Todas aquellas tj de las que ti ha leído algún elemento.
El protocolo **2V2PL (bloqueo de dos versiones)** mantiene como máximo dos versiones de cualquier elemento de datos en cada
momento. Supongamos que ti escribe el elemento de datos x, pero aún no está comiteado, las dos versiones de x son su imagen anterior y su imagen posterior. Tan pronto como ti se comitea, la imagen anterior puede ser eliminada ya que la nueva versión de x ahora es estable, y las versiones antiguas ya no son necesarias ni se mantienen. En 2V2PL las operaciones de lectura están restringidas a leer solo las versiones actuales, es decir, la última versión comiteada.
Locks rl read lock: que se establece antes de una operación de lectura r (x) respecto de la versión actual de x **wl write lock**: que se establece antes de una operación de escritura w (x) para escribir una versión no commiteada de x.**cl commit o certify lock**:se establece un cl(x) antes de la ejecución del paso final de una transacción en cada elemento de datos x que esta transacción ha escrito.Las operaciones de unlock deben obedecer al protocolo 2PL. cl no es compatible con ningún otro lock.
Logging:
Transaccion incompleta sii no hay ni commit ni abort record
UNDO Logging Forma de los Update Record: < Ti , X, v > La transacción Ti actualizó el valor del registro X que antes valía v **Reglas**: 1. Cada Update Record < Ti , X, v > se baja a disco antes de bajar a disco el nuevo valor de X 2. Cada Commit Record < COMMIT Ti > se baja a disco después de bajar a disco todos los cambios realizados por Ti 3. Una vez que los cambios estén asegurados en el disco, registrar en el log que esos cambios son válidos (commit) 4. Puede haber cambios en el disco que aún no haya registrado como válidos (commit) en el log.
Recuperacion: Idea: Identificar las transacciones incompletas y deshacer los cambios que realizaron en el disco 1. Recorrer el log desde el final hacia atrás 2. Para cada transacción Ti , recordar si se encuentra un Commit Record o un Abort Record 3. Por cada Update Record < Ti , X, v >, si Ti es una transacción incompleta hay que deshacer el cambio: Asignar v a X 4. Por cada Ti incompleta agregar un Abort Record al log y bajarlo al disco
REDO Logging Forma de los Update Record: < Ti , X, v > La transacción Ti actualizó el valor del registro X que pasa a valer v **Reglas**:1. Cada Update Record < Ti , X, v > se baja a disco antes de bajar a disco el nuevo valor de X 2. Cada Commit Record < COMMIT Ti > se baja a disco antes de bajar a disco cualquiera de los cambios realizados por Ti 3. Primero asegurar los cambios en el log (commit), luego escribirlos en el disco 4.Las actualizaciones en el log no necesariamente quedaron reflejadas en el disco.
Recuperación: Idea: Identicar las transacciones completas y rehacer los cambios que no llegaron a bajarse al disco 1.Recorrer el log una primera vez para identicar cuáles son las transacciones completas y cuáles las incompletas 2. Recorrer el log una segunda vez, desde el principio hacia adelante 3. Por cada Update Record < Ti , X, v >, si Ti es completa y hay que actualizar el
cambio en el disco: Asignar v a X 5. Por cada Ti incompleta agregar un Abort Record al log y bajarlo al disco
UNDO/REDO Logging: Forma de los Update Record: < Ti , X, v0 , v1 > La transacción Ti actualizó el valor del registro X que antes valía v0 y pasa a valer v1
Reglas:1. Cada Update Record < Ti , X, v0 , v1 > se baja a disco antes de bajar a disco el nuevo valor de X 2. Cada vez que se va a cambiar algo en disco, primero grabar el registro de dicho cambio en el disco
Recuperación: Aplicar UNDO a las transacciones incompletas en orden inverso 1. Recorrer el log desde el final hacia atrás 2. Para cada transacción Ti , recordar si se encuentra un Commit Record o un Abort Record 3. Por cada Update Record < Ti , X, v0 , v1 >,si Ti es incompleta y hay que deshacer el cambio: Asignar v a X **Aplicar REDO** a las transacciones comiteadas en orden
1. Recorrer el log una segunda vez, desde el principio hacia adelante 2. Por cada Update Record < Ti , X, v0 , v1 >, si Ti es completa y hay que actualizar el cambio en el disco: Asignar v a X **Por cada Ti incompleta** agregar un Abort Record al log y bajarlo al disco
Checkpoint Quiescente con UNDO Logging Etapas: 1. Dejar de aceptar nuevas transacciones 2. Esperar a que todas las transacciones activas (aquellas con Start Record pero sin Commit ni Abort Record) comitéen o aborten 3. Agregar el registro < CKPT > al log y bajarlo a disco 4. Volver a aceptar nuevas transacciones
Recuperación: Aplicar UNDO desde el final del log hasta el último checkpoint (el primero encontrado)
Checkpoint No-Quiescente con UNDO Logging Etapas:1. Agregar el registro < START CKPT(T1 , T2 , ..., Tk) > al log y
bajarlo al disco (siendo T1 , T2 , ..., Tk las transacciones activas) 2. Esperar a que todas las transacciones activas comitéen o aborten, pero sin restringir que empiecen nuevas transacciones 3. Agregar el registro < END CKPT > al log y bajarlo a disco
Recuperación: 1. Aplicar UNDO desde el final del log 2. Si se encuentra un < END CKPT >, seguir hasta el < START CKPT(...) > 3. Si se encuentra un < START CKPT(T1 , T2 , ..., Tk) >, seguir hasta el < START Ti > más antiguo, con 1 ≤ i ≤. O sea, las que quedaron incompletas.
Checkpoint No-Quiescente con REDO Logging Etapas: 1. Escribir < Start CKPT(T1 , T2 ...Tk) > en el log, y efectuar un flush. T1 , T2 ...Tk son las transacciones activas al momento de introducir el checkpoint. 2. Forzar a disco los ítems que fueron escritos en el pool de buffers por transacciones que hicieron commit al momento del < Start CKPT >. pero que aún no fueron guardados en disco. 3. Escribir < End CKPT > en el log y efectuar un flush del mismo.
Recuperación: 1. Identicar el último checkpoint que finalizó correctamente (el último < Start CKPT(...) > que tiene un < End CKPT > posterior) 2. Si se encuentra un < Start CKPT >. (no hay < End CKPT > debido a un crash), ubicar el < End CKPT > anterior y su correspondiente < Start CKPT(T1 , T2 ...Tk) > y rehacer todas las transacciones comiteadas que comenzaron a partir de ahí o están entre las Ti .
Checkpoint No-Quiescente con UNDO/REDO Logging Etapas: 1. Escribir < Start CKPT(T1 , T2 ...Tk) > en el log, y efectuar un flush. T1 , T2 ...Tk son las transacciones activas 2. Escribir a disco todos (no sólo los que hayan sido modicados por transacciones comiteadas) los buffer sucios al momento del start
Recuperación: 1. **Transacciones incompletas**: para deshacerlas se debe retroceder hasta el start más antiguo de ellas. Agregar registro < ABORT Ti > al log para cada transacción Ti incompleta, y hacer flush del log 2. **Transacciones con commit** : Si leyendo desde el último registro se encuentra un < End CKPT > sólo será necesario rehacer las acciones efectuadas desde el correspondiente registro < Start CKPT(T1 , T2 ...Tk) > en adelante. Si no encontramos el < End CKPT >, usamos la estrategia del REDO Redo
Key Value: Características 1. Simples 2. Escalables 3. Veloces. **Clave:** Entity Name + ':' + Entity Identifier + ':' + Entity Attribute. Usar nombres significativos y no ambiguos Usar un delimitador común ":" Mantener las claves lo más cortas posibles sin sacrificar las otras características. **Valores:** String, Lista, Hash, JSON
Consistencia Estricta/Fuerte Un sistema tiene consistencia estricta cuando las lecturas retornan siempre el valor mas reciente del ítem que se lee
Consistencia Débil Existe consistencia débil cuando el sistema no garantiza que una lectura obtenga un valor actualizado.
Consistencia Eventual Las operaciones de lectura verán conforme pasa el tiempo las escrituras En un estado de equilibrio el sistema devolvería el último valor escrito. A medida que t− > ∞ los clientes verán las escrituras.
Consistency: Todos ven los mismos datos al mismo tiempo **Availability**: Si se puede comunicar con un nodo en el cluster entonces se pueden leer y escribir datos.
Partition tolerance: El cluster puede sobrevivir a roturas de comunicación que lo dividan en particiones que no pueden comunicarse entre ellas.



Document database: Las BD almacena y recupera documentos que se agrupan en colecciones.
Cassandra (Wide column store): Las columnas de la clave de partición solo admiten dos operadores: = e IN. Cassandra requerirá que restrinja todas las columnas de clave de partición o ninguna de ellas a menos que la consulta pueda usar un índice secundario. **Se crea una tabla por consulta.**
MR1 (Entities and Relationships): Los tipos de entidades y relaciones mapean a tablas mientras que los datos se asignan a filas. Los atributos de las entidades y las relaciones se mapean a columnas **MR2** (Equality Search Attributes): Si se utilizan en una consulta por igualdad de atributos, entonces, éstos se mapean a columnas del prefijo de la clave primaria. Dichas columnas deben incluir todas las columnas de clave de partición y, opcionalmente, una o más columnas clustering key **MR3** (Inequality Search Attributes): Si se utilizan en consultas por desigualdad, estos atributos mapean como columnas clustering key. En la definición de clave principal, una columna que participa en la búsqueda de desigualdad debe ubicarse después de las columnas que participan en la búsqueda de igualdad. **MR4** (Ordering Attributes): Mapea a una columna clustering key con orden ascendente o descendente según se especifique en la consulta **MR5**(Key Attributes): Mapea a columnas en la clave primaria. Una tabla que almacena datos de entidades o relaciones como filas debe incluir atributos claves que identifique estos datos unívocamente