Las Restricciones de Integridad protegen a la base de datos de daños accidentales, no permitiendo datos que resulten en un estado inconsistente. Una de las ventajas de mantener la información con un DBMS es la posibilidad de garantizar ciertos niveles de integridad de datos. De esta forma los DBMS proveen mecanismos para implementar restricciones de integridad. Entre estos mecanismos se encuentran:

- Claves: el conjunto de instancias legales está restringido a que dos o más tuplas no compartan los mimos valores para los atributos claves.
- Multiplicidad de la relación: restringe la cantidad de relaciones legales entre entidades.
- Dependencias Funcionales.
- Restricciones de dominio: ajustar el dominio de los atributos de manera tal que se restringe cada atributo a un dominio de valores posibles. SQL ofrece varios mecanismos NOT NULL, PRIMARY KEY, UNIQUE, CHECK(P).
- Restricciones de Integridad Referencial: aseguran que el valor que aparece en una relación para un conjunto de atributos este presente en otra relación para cierto conjunto de atributos. Dados r(R) y s(S) puede darse que al hacer r|><|s una tupla t de r no haga join con ninguna de s, así t sería colgante. No siempre es correcto que existan tuplas colgantes. Esto se logra mediante el uso de claves foráneas.
- Triggers: es una orden que ejecuta el sistema como efecto de un cambio (ON evento IF precondición THEN acción).

Transacción: colección de operaciones sobre una base de datos que forman una unidad lógica de trabajo. Eventualmente puede acceder y/o actualizar varios ítems de datos. Accede y deja a la DB en un estado consistente. Múltiples transacciones se pueden ejecutar en paralelo. Es el programador quién define los límites de una transacción. EL DBMS debe asegurar la ejecución adecuada de cada transacción, a pesar de la existencia de fallos.

Propiedades ACID

- **Atomicity (DBMS):** Las operaciones de una transacción se ejecutan todas o ninguna. Un ejemplo sería una transacción que se aborta sin terminar.
- **Consistency (Programmer):** La ejecución de una transacción en aislamiento preserva la consistencia de la BD.
- **Isolation (DBMS):** Aunque varias transacciones se ejecuten de forma concurrente, el resultado debe ser equivalente a alguna secuencia de ejecuciones en serie.
- **Durability (DBMS):** Una vez ejecutada con éxito la transacción, los cambios en la BD persisten, más allá de fallas en el sistema. Para garantizar esto hay que hacer que las modificaciones se escriban en disco antes que la transacción finalice o la información sobre los cambios efectuados por la transacción es suficiente para repetirla en caso de falla.

Planificación: la secuencia cronológica en la cual se ejecutan en el sistema las instrucciones de transacciones concurrentes.
Una planificación se dice en **serie** si las instrucciones que pertenecen a una misma transacción aparecen juntas. Una planificación, posiblemente concurrente, se dice **serializable** si el resultado de su ejecución es equivalente al de alguna planificación en serie.
Conflicto: sean las instrucciones Ii y Ij referentes a dos transacciones Ti y Tj

respectivamente. Se dice que están en conflicto cuando son instrucciones de transacciones distintas sobre el mismo dato y al menos una de ellas es una instrucción Write.
Si Ii e Ij no están en conflicto, entonces se pueden intercambiar para obtener una planificación Ŝ equivalente a S. S y Ŝ son equivalentes puesto que las instrucciones aparecen en el mismo orden salvo para las instrucciones Ii e Ij cuyo orden no importa.

Existen dos formas de equivalencia entre planificaciones:

Seriabilidad en conflictos: una planificación S es **serializable en cuanto a conflictos** si es equivalente en conflictos a una planificación en serie. Para esto se hace el test de grafo de procedencia, se crea un grafo dirigido donde cada vértice corresponde con una transacción. Se agrega un arco de Ti a Tj si:

- Ti hace write(Q) antes que Tj hace read(Q).
- Ti hace read(Q) antes que Tj hace write(Q).
- Ti hace write(Q) antes que Tj hace write(Q).

De existir un ciclo en el grafo, entonces la planificación no es serializable en conflictos.

Seriabilidad en vistas: una planificación es S es equivalente en vistas a una Ŝ si cumple con las tres condiciones siguientes para cada ítem de dato Q:

1. Si Ti ejecuta read(Q) y lee el valor inicial de Q en S, entonces Ti debe valer el valor inicial de Q en Ŝ.
2. Si Ti ejecuta read(Q) en S y ese valor fue producido por Tj (si existe), entonces Ti en Ŝ debe leer el valor producido por Tj.
3. La transacción, si existe, que ejecuta el write(Q) de la planificación S debe ejecutar la operación final write(Q) en la planificación Ŝ.

1 y 2 aseguran que cada transacción lea los mismos valores en ambas planificaciones, y por lo tanto, realicen el mismo cálculo. 3, junto con 2 y 1, aseguran que ambas resulten en el mismo resultado final.

Una planificación es serializable en vistas si es equivalente en vistas a una planificación en serie. La equivalencia en vistas es menos rigurosa que la de conflictos.

Los módulos para testear si una planificación es serializable en vistas tienen un costo exponencial relativo al tamaño del grafo de precedencia. El problema de chequear si es serializable en vistas cae en la clase de problemas NP-completos, por lo que la existencia de un algoritmo eficiente es poco probable.

Básicamente, toda planificación serializable en conflictos es también serializable en vistas. A su vez, toda planificación serializable en vistas es además serializable.

En entornos multiprogramados es posible ejecutar varias transacciones en forma concurrente. Las ventajas del procesamiento concurrente son mejor utilización de disco y procesador (throughtput) y mejora de los tiempos de respuesta, dado que transacciones cortas no necesitan esperar por transacciones largas.

La concurrencia puede llevar a la pérdida de consistencia aun cuando la lógica de las transacciones individuales sea correcta. Los **esquemas de control de concurrencia** son los mecanismos para lograr **aislamiento** (Isolation). Estos permiten que se generan planificaciones concurrentes, pero asegurando planificaciones serializables. Los protocolas

imponen reglas que evitan planificaciones no serializables. Los distintos protocolos varían en cuanto al paradigma, el nivel de concurrencia y el overhead.

Si una transacción falla durante su ejecución, para asegurar atomicidad, se deben deshacer sus efectos. En un sistema concurrente, si existe una transacción Tj que depende de Ti, y Ti falla, Tj también deberá ser retrocedida.

Planificación no recuperable: si existe Tj que depende de Ti, pero Tj comete antes que Ti termine y posteriormente Ti falla.

Retroceso en cascada: si existe Tk que depende de Tj, Tj que depende de Ti.... Si falla Ti deberán retroceder en cascada Ti, Tj y Tk.

Protocolo Basado en Bloqueos

Un lock es un mecanismo para controlar el acceso concurrente a un item de dato. Hay dos modos de bloqueo:

- Compartido (lock-S): permite la lectura sobre Q, pero no la escritura.
- Exclusivo (lock-X): si T ha obtenido un bloque exclusivo sobre Q, entonces podrá leer y escribir sobre Q.

Las transacciones envían sus pedidos de bloqueo al gestor de control de concurrencia quien concederá su solicitud si es compatible con los bloqueos ya asignados a otras transacciones sobre el mismo dato. Básicamente, se da el bloqueo compartido si había un bloque compartido sobre Q, si hay un bloqueo exclusivo previo se deberá esperar. El bloqueo exclusivo sólo se da si no hay bloqueos previos. En general se mantiene una tabla de bloqueos implementada como una tabla hash en memoria.

El problemas principal es que no garantiza serializabilidad. Además, no está libre de deadlock ni inanición. Es por esto que no puede usarse como protocolo de control de concurrencia.

Protocolo de bloqueo de dos fases

Requiere que cada transacción realice sus solicitudes de bloqueo y desbloqueo en dos fases. La primera de Crecimiento, donde sólo puede pedir y no liberar ningún bloqueo. La segunda de Encogimiento, donde se pueden liberar bloqueos pero no pedirlos. Este protocolo asegura una planificación serializable en conflictos y el orden de serializabilidad de las transacciones está dado según su punto de lock, el último obtenido. No está libre de deadlocks. Puede generar retrocesos en cascada y planificaciones no recuperables como resultado de un retroceso en cascada.

Refinamientos a PB2F: permitir conversiones de bloqueos. Esto es pasar de un bloqueo lock-S a lock-X sobre Q en la fase de crecimiento (upgrade). Pasar de bloqueo lock-X a lock-S en la fase de encogimiento (downgrade).

Protocolo de bloqueo de dos fases estricto: Similar al PB2F sólo que todos los bloqueos exclusivos se conservan hasta que se comete la transacción. Soluciona el problema de retrocesos en cascada y planificaciones no recuperables.

Protocolo de bloqueo de dos fases riguroso: Exige que toda transacción mantenga todos sus bloqueos hasta el momento que se comete. El orden de serializabilidad es el orden en que se cometen las transacciones.

Protocolo basado en grafos

Imponen un orden parcial sobre el conjunto D de ítems de datos. Dado D={d1, d2, ...dn}, si en D di->dj entonces cualquier transacción que necesite acceder a los datos i y j deberá hacerlo en ese orden.

Tiene un único tipo de bloqueo permitido, lock-X. El primer bloqueo puede ser sobre

cualquier dato, luego, podrá bloquear un dato Q solamente si se tiene el bloqueo del padre. Se pueden desbloquear los datos en cualquier momento, pero una vez desbloqueado no se puede volver a bloquear.

Asegura serializabilidad de conflictos y está libre de deadlocks. No está libre de retrocesos en cascada y planificaciones no recuperables. Requiere el bloqueo de datos que no se necesitan, lo que disminuye la concurrencia.

Protocolo basado en hora de entrada (time-stamp)

Se resuelven los conflictos realizando un esquema de ordenamiento según la hora de entrada o inicio de la transacción. Cuando Ti ingresa, se le asigna una hora ts(Ti); si luego ingresa Tj tendremos que ts(Ti)<ts(Tj).

Este esquema se puede implementar con el reloj de sistema o un contador lógico. En los protocolos basados en hora de entrada se manejan las ejecuciones concurrentes de manera que las estampillas de tiempo de las transacciones determinan el orden de la serializabilidad. Si ts(Ti)<ts(Tj) entonces el sistema debe asegurar que la planificación sea equivalente a una planificación en serie en la cual Ti precede a Tj.

Cada dato Q requiere así dos estampillas de tiempo:

•R-ts(Q): mantiene la mayor etiqueta de tiempo de una transacción que ejecutó exitosamente un read(Q).

•W-ts(Q): mantiene la mayor etiqueta de tiempo de una transacción que ejecutó exitosamente un write(Q).

Dada la situación en que Ti desea realizar un **read(Q)**, el control consistirá en:

- **ts(Ti)<W-ts(Q):** si Ti necesita leer un valor Q que ya fue escrito, la operación es **rechazada**.
- **ts(Ti)>=W-ts(Q):** se ejecuta el read **exitosamente**, y el nuevo valor de R-ts(Q) será el máximo entre ts(Ti) y el previo valor de R-ts(Q).

Dada la situación en que Ti desea realizar un **write(Q)**, el control consistirá en:

- **ts(Ti)<R-ts(Q):** el valor que produce Ti fue requerido previamente y el sistema asumió que nunca se produciría, por lo que la operación es **rechazada**.
- **ts(Ti)<W-ts(Q):** entonces Ti está intentando escribir un valor obsoleto de , por lo que es **rechazada**.
- En cualquier otro caso, el write se ejecuta con éxito y W-ts(Q) pasa a ser ts(Ti).

Este esquema asegura que las instrucciones read y write en conflicto se ejecuten en le orden de entrada, garantiza seriabilidad en conflictos. Esta libre de deadlocks pues las transacciones nunca esperan. No está libre de retrocesos en cascada ni de planificaciones no recuperables.

Con este protocolo existen casos de retrocesos innecesarios cuando Ti intenta escribir un valor obsoleto. La Regla de Escritura de Thomas dice que **si ts(Ti)<W-st(Q) la operación de escritura puede ser ignorada.**

Control de Concurrencia Pesimista: realiza una ejecución cauta de las tareas, cada acceso a la base de datos es cuidadosamente verificado, tomando las acciones apropiadas en ese preciso instante. No se requiere una segunda etapa de validación, ya que cada operación es verificada antes de hacerse efectiva. La desventaja es que el chequeo sobrecarga el sistema y demora la ejecución de las transacciones. Ej. de estas son PB2F y sus variantes, Protocolo Árbol, Estampilla de Tiempo, Estampilla de Tiempo+Regla de escritura de Thomas, Multiversión.

Control de Concurrencia Optimista: asume condiciones que simplifican el

desarrollo de la tarea, luego en una segunda etapa realiza la validación para chequear si las condiciones asumidas fueron ciertas. De no serlo, deberá que retroceder la transacción y ejecutarla nuevamente. No existe posibilidad de deadlock, dado que ninguna transacción espera por otra.

Protocolo Basado en Validación

Es útil y provee mayor grado de concurrencia en la medida que las probabilidades de conflictos entre transacciones concurrentes sea baja. Previene el retroceso en cascada.

Consiste en 3 fases:

1. Lectura: tiene lugar la ejecución de Ti, se los lo valores de los diferentes datos y se almacenan las variables locales.

2. Validación: se determina si se puede copiar a la base de datos, sin violar serializabilidad, las variables locales temporales.

3. Escritura: se aplican las actualizaciones reales a la base de datos, de haber pasado la validación. Caso contrario, retrocede.

Cada transacción Ti tendrá 3 time stamps:

- Start(Ti): con la hora que comenzó su ejecución.
- Validatio(Ti): la hora en la que comenzó su validación.
- Finish(Ti): la hora en que terminó su fase de escritura.

El orden de seriabilidad está determinado por las estampillas de validación. La prueba de validación para Tj requiere que para cada transacción Ti con ts(Ti)<ts(Tj), es decir Ti previo a Tj, se debe cumplir **una** de las siguientes condiciones:

1. **Fin(Ti)<Start(Tj):** Ti terminó antes que Tj empezara.
2. El conjunto de **datos que escribe Ti no tiene intersección con los que lee Tj** y Ti termina su escritura antes que Tj empiece su validación, es decir, **Start(Tj)<Fin(Ti)<Validate(Tj)**. Esto garantiza que las escrituras no se solapen y se mantenga el orden.

Esquema Multiversión

Cada write(Q), si es exitoso, crea una nueva copia de Q mientras que en cada operación read(Q) el sistema selecciona que copia es leída. Con cada ítem de dato Q se asocia una secuencia de versiones <Q1, Q2,..., Qn>. Para cada versión Qk se tienen tres ítems de datos:

- el contenido de la versión de Q.
- R-ts(Q): la mayor estampilla de tiempo de una transacción que leyó a Q.
- W-ts(Q): la estampilla de tiempo que creó la versión de Q.

Un write exitoso crea una nueva versión de Q, donde los valores de W-ts y R-ts serán ts(Ti). El valor de R-ts se actualiza si una transacción Tj lee el contenido de Qk y R-ts(Qk)<ts(Tj).

Supongamos que una transacción Ti realiza una operación read o write. Sea **Qk la versión de Q cuya estampilla de tiempo es la más grande menor o igual a ts(Ti)**. Si Ti realiza un read, entonces el valor retornado es Qk. Si realiza un write, en cambio, habrá tres casos:

1. si **st(Ti)<R-ts(Q)**, entonces la transacción **retrocede**.
2. Si **ts(Ti)=W-ts(Q)**, entonces se **sobreescribe** Q.
3. Se crea una **nueva versión** de Q.

Las versiones que no se usan más pueden borrarse si el W-ts es menor que la estampilla de tiempo de la transacción más antigua del sistema. Las lecturas nunca fallan y nunca esperan. Causa menos abortos que otros esquemas, aunque los conflictos se

resuelven por retrocesos, no por esperas. Leer un dato implica actualizar su hora, por lo que requiere dos accesos a disco en vez de uno. El DBMS debe descubrir cunado una versión es inaccesible para poder eliminarla. Funciona muy bien en situaciones de lectura solamente. Si la cantidad de conflictos es alta los protocolos basados en bloqueos se comportan mejor.

El DBMS deberá o bien prevenir los deadlocks, o dejar que ocurran y accionar para solucionarlos. Un sistema está en estado de deadlock si existe un conjunto de dos o más transacciones tal que toda transacción del conjunto está esperando por un dato retenido por otra transacción del conjunto.

Estrategias:

1. Algoritmo del banquero.
2. Imponer un orden parcial y exigir que se respete (árbol).
3. Combinar bloqueos con estampillas de tiempo: las transacciones que solicitan un bloqueo de dato Q y que pueden generar deadlock son retrocedidas según:
   - **Wait-Dies:** Ti requiere un dato que tiene Tj. Si Ti es más antigua que Tj, Ti espera. Sino es retrocedida. Así una transacción antigua puede esperar bastante antes de obtener su dato.
   - **Wound-Waits:** Ti requiere un dato que tiene Tj. Si Ti es más joven que Tj espera. De lo contrario Tj es retrocedida y Ti se apropia del dato. Puede tener menos retrocesos que Wait-Die. Además, las transacciones más viejas tienen precedencia sobre las nuevas y de esta manera se evitan problemas de inanición.
4. Time-out: se permite el bloque un tiempo determinado, pasado el límite la transacción debe retroceder. Previene deadlocks y es simple de implementar. Tiene posibilidades de inanición y es difícil determinar el tiempo de timeout.

Granularidad para Q

¿Cuál es la granularidad del ítem de dato a bloquear? Base de datos, Tablas, páginas o registros. La granularidad fina permite alta concurrencia pero alto overhead por bloqueos. La granularidad gruesa tiene un pequeño overhead por bloqueos pero la concurrencia es baja.

Las ventajas del bloqueo a nivel registro son los menores conflictos si las transacciones acceden a filas distintas, por lo que se puede mantener un lock por más tiempo. La desventaja es que la tabla de bloqueos requiere más memoria y es más lenta si una operación requiere de varios bloqueos (ej. group by). Los bloqueos a nivel página son apropiados si la mayoría de operaciones son lectura y hay muchos accesos usando group by.

Se puede permitir granularidad múltiple al permitir que coexistan bloqueos en unidades distintas entre diversas transacciones. El modo bloqueo, ahora, alcanza al nodo y sus descendientes. Para hacerlo práctico se emplean tipos adicionales de bloqueos llamados bloqueos de intención.

Intention Share (IS): en algún nivel inferior existen bloqueos explícitos compartidos.

Intention Exclusive (IX): en algún nivel inferior existen bloqueos explícitos exclusivos o compartidos.

Shared Intention Exclusive (SIX): el subárbol está bloqueado explícitamente en modo compartido y algún hijo en modo exclusivo.

El problema del gestor de bloqueos es manejar los bloqueos implícitos.

Todo sistema está sujeto a fallos,un fallo puede generar pérdida o inconsistencia de información. Ante un fallo el DBMS debe recuperarse y dejar la base de datos en un estado consistente anterior al fallo. Los fallos se clasifican en:

1. **Fallo de Transacción**: afecta a la transacción, esta deberá retroceder. El alcance del fallo de la transacción afecta a la transacción activa y potencialmente a otras en efecto cascada. Se dividen en:
   - **Error lógico**: la transacción no puede continuar su ejecución debido a alguna condición interna.
   - **Error bajo el entorno del sistema**: el sistema debe terminar la ejecución de una transacción activa debido a alguna condición de error, la transacción vuelve a ejecutarse más tarde.
2. **Caída del sistema (Crash)**: fallos del Hw o del Sw que causan la falla general y caída del sistema. Causan pérdida de información en memoria volátil (Ppal. Y cache). El contenido de información en memoria no volátil permanece intacto. El DBMS posee numerosos chequeos de consistencia para prevenir corrupción de datos. El fallo afecta a todas las transacciones activas al momento del error y eventualmente a transacciones recientemente cometidas.
3. **Falla de Disco**: daño físico en el medio de almacenamiento masivo. Se requiere de copias de datos de backup en otros discos u otros medios de almacenamiento para recuperar el fallo. La recuperación combina los pasos de recuperar un backup con recuperación del sistema.

Almacenamiento Volátil: memoria principal y cache, no sobrevive a las caídas de sistema. Tiempos de acceso superiores.

Almacenamiento no Volátil: cintas magnéticas y discos. Sobrevive a las caídas de sistema.

Almacenamiento Estable: medios de almacenamiento que aseguren que la información nunca se pierde. Sobrevive a todos los fallos, replica la información en varios medios con modos de fallos independientes manera controlada.

Sistemas de recuperación

Los algoritmos de recuperación son técnicas para asegurar **consistencia** y **durabilidad** de las transacciones a pesar de fallos. En un AR se identifican acciones preventivas a tomar durante el procesamiento normal de las transacciones para asegurar que exista suficiente información para permitir la recuperación. Existen además, acciones de recuperación como consecuencia de un fallo para asegurar la consistencia, atomicidad y durabilidad de las transacciones.

Las **bitácoras**, log, se mantienen en **memoria estable** y contiene la secuencia de registros con información sobre las actualizaciones realizadas sobre la base de datos. Los registros de la bitácora deberán ser escritos siempre antes que la modificación a la DB. Cuando una transacción Ti se inicia se agrega a la bitácora: <Ti, start>. Cuando ejecuta un write(x) se registra la transacción, el dato, el valor viejo y el nuevo: <Ti, x, vi, vf>. Cuando la transacción finalizó con su última instrucción agrega: <Ti, commit>, de abortar registra <Ti, abort>.

El Undo de un registro <Ti, X, V1, V2> escribe el valor viejo V1 en X. El Redo de un registro <Ti, X, V1, V2> escribe el valor nuevo V2 en X.

Esquema de Modificación Inmediata

Las modificaciones sobre las DB se realizan en cualquier momento aún mientras la transacción está activa. En caso de ocurrir un fallo, se deberán deshacer los cambio hechos por una transacción no cometida. Este esquema hace uso de Undo y Redo. Básicamente se genera una lista de operaciones a deshacer de aquellas transacciones que no hicieron el commit a la hora de fallo, luego se deshacen en orden inverso al que se ejecutaron inicialmente. Luego, se hace el redo de todas aquellas que tienen un start y un commit.

Esquema de Modificación Diferida

Implica atomicidad de la transacción grabando las modificaciones en la bitácora y aplazando las operaciones write en la base de datos hasta que la transacción este parcialmente cometida. La información de la bitácora se utiliza cuando se ejecutan escrituras diferidas. Si ocurre un fallo de sistema o de transacción antes de que termine de ejecutarse, se ignora la información de la bitácora. Si se realiza una operación write el registro será de la forma <Ti, X, V>, el valor viejo no importa. Para recuperar une estado consistente ante una caída de sistema, se deberá hacer redo de todas aquellas transacciones que tienen start y commit en la bitácora.

Ante un fallo de sistema las acciones a seguir son: iniciar el proceso de recuperación, suspender la atención de requerimientos de usuarios, analizar las bitácoras determinando el conjunto de transacciones a rehacer y deshacer, reiniciar la operación normal del sistema. Cada vez que ocurre un fallo de sistema sería necesario recorrer la bitácora completa, lo que consume tiempo.

La solución a esto son los Checkpoints, puntos de verificación que graban en disco todos los registros de bitácora que están en memoria principal, graban en disco los bloques modificados de los registros intermedio y graba un registro en bitácora <checkpoint> en memoria. Así, toda transacción que tenga su commit antes del checkpoint en la bitácora, no requerirá redo. Para restaurar el sistema, sin embargo, se deberá buscar hacia atrás el primer checkpoint y el start más cercano (en un sistema en serie), luego se aplica undo y redo sobre esa transacción y todas las que la suceden.

En un sistema concurrente, <checkpoint> pasa a ser <checkpoint L> donde L es una lista de las transacciones activas al momento de checkpoint. La lista L limitará la búsqueda hacia atrás del checkpoint. Primero se recorre la bitácora hacia atrás hasta el primer checkpoint, por cada commit luego del checkpoint se agrega la transacción a la lista redo. Por cada registro de una transacción start, si no está en redo se inserta en la lista undo. Al llegar al checkpoint se controla L, por cada transacción que está en L pero no en la lista redo, se agrega a la lista undo.

Una vez que se tienen las listas se recorre la bitácora hacia atrás, haciendo undo de cada registro cuya transacción está en la lista undo. Del último checkpoint de la bitácora hacia abajo, se irá realizando redo de cada registro que corresponda a una transacción en la lista redo.

Sistema Centralizado: incluye una, o pocas, CPU, una memoria principal y una serie de controladores conectados a través de un bus común que provee acceso a memoria compartida. EL CPU y los controladores pueden ejecutar acciones de manera concurrente, compitiendo por el acceso a memoria. Se distinguen en sistemas mono-usuarios y multi-usuarios (terminales).

Los mono-usuario son usados y administrados por una persona al mismo tiempo, no proveen control de concurrencia. Los multi-usuarios atienden varios usuarios operando el sistema al mismo tiempo, disponen de mayor capacidad de disco, memoria y CPU; disponen de facilidad para multitareas.

Sistemas Paralelos: consisten de múltiples procesadores y múltiples discos conectados por una red de alta velocidad. Mejoran el procesamiento y la velocidad de I/O. En estos muchas operaciones se desarrollan en paralelo. Surgen con la demanda de aplicaciones que manejaban las bases de datos muy grandes y de miles de transacciones por minuto. Existen varios modelos de arquitectura para máquinas paralelas:

memoria compartida: eficiente en la comunicación entre procesadores, el bus resulta un cuello de botella ante un gran número de procesadores.

Disco compartido: el bus no es cuello de botella, proporciona tolerancias a falla del procesador, el cuello de botella está en la conexión a disco.

Nada compartido: son ampliables y pueden soportar un gran número de procesadores, minimiza las interfaces por recursos compartidos, el costo de comunicación y acceso a discos no locales es muy grande.

Jerárquico (una combinación de los anteriores).

Sistemas Distribuidos: la base de datos se almacena en varias computadoras, denominadas sitios o nodos, comunicadas entre sí. Los nodos están en servidores que no comparten memoria ni disco y se conectan por la red. Las computadoras pueden variar en tamaño y función, y estar ubicados en diferentes lugares. Las transacciones pueden acceder a datos que están físicamente en sitios distintos. Los sistemas distribuidos se asemejan a los paralelos con nada compartido, sin embargo, suelen estar en lugares diferentes, son administrados independientemente y tienen una conexión más lenta. En un sistema distribuido surgen más fallas: falla de un sitio, pérdidas de mensajes, fallo físico de un enlace, partición de la red (uno o más nodos están aislados del sistema).

- DB Distribuidas Homogéneas: el mismo Sw/esquema está en todos los nodos, los datos son particionados/replicados por todos los nodos. Provee una vista de una sola base de datos ocultando detalles de distribución. Cada nodo sabe de la existencia de los otros y están de acuerdo en cooperar con la atención de los requerimientos de usuarios. Cada sitio sacrifica autonomía para conservar un esquema idéntico. Proveen además un entorno para ejecutar tanto transacciones locales como globales.
- DB Distribuidas Heterogéneas: diferentes Sw/esquemas en los nodos. Su objetive es integrar distintas bases de datos para proveer una función útil. Los sitios pueden no saber de la existencia de los otros. Permite preservar la identidad original y la selección del DBMS.

Cada sitio cuenta con un **gestor de transacciones** que gestiona la ejecución de aquellas **transacciones que acceden a los datos almacenados en el sitio local**, además deberá mantener la bitácora y controlar la concurrencia. Tienen también un **coordinador de transacciones** encargado de ejecutar las **transacciones locales y globales iniciadas en el sitio local**.

El sistema distribuido tiene las ventajas de permitir a los usuarios de un sitio acceder a los datos de otro sitio. Brinda además autonomía, cada sitio es capaz de mantener el control de los datos que están almacenados localmente. Existe un administrador global del sistema, y cada sitio cuenta con un administrador local. Da además mayor disponibilidad, de fallar un sitio, el resto puede continuar operando.

Las desventajas son claras, el desarrollo de sus Sw es caro, hay más probabilidad de errores y hay un incremento en la sobrecarga del procesamiento (mensajes).

El almacenamiento de los datos en un sistema distribuido se puede dar por:

- Replicación: se mantienen varias copias idénticas en sitios diferentes, esto mejora el tiempo de respuesta y la tolerancia a fallos. Incrementa el overhead de las actualizaciones e incrementa la complejidad del control de concurrencia.
- Fragmentación Vertical (por columnas): una relación se divide por atributos, dividir a la relación para proteger campos. Permite distribuirla de manera tal que cada parte de la tupla se encuentre donde más se utiliza. Permita procesamiento paralelo sobre los campos de una relación.
- Fragmentación Horizontal (por filas): se separa una relación dividiendo las tuplas en subconjuntos. Permite procesamiento paralelo sobre fragmentos de la relación. Permite que cada tupla se distribuya donde más frecuentemente se utiliza.

En un DDBMS se definen dos conceptos para dato: el lógico y el físico. Si un dato A tienen n copias, se dice que A es el dato lógico y Ai es el dato físico. En un sistema distribuido con replicación el bloqueo debe darse a nivel de dato lógico.

Gestión de Bloque Centralizado: Nodo Centralizado

Nodo Centralizado: Se define un gestor de bloques en el sitio S. Si se desea bloquear el dato Q, se envía el requerimiento a S que determina si puede concederlo. Es simple de implementar, y el control de deadlock es simple. El lock manager será un cuello de botella y es vulnerable a fallos.

Gestión de Bloques Distribuida: ROWA, Mayoría, K de n

ROWA (Read-Lock-One Write-Lock-All): Es sencillo, para obtener un read-lock sobre A deberá tener un lock sobre una copia de A, (un mensaje de control y 1 de datos, la concesión del bloqueo va con el dato). Para obtener un write-lock, se deberá tener lock sobre todas las copias (requiere 2n mensajes de control, n para el pedido y n para la liberación, y n mensajes de datos). Funciona bien en entornos donde predominan las lecuras. Puede haber deadlock debido al pedido de escritura.

Mayoría: Para obtener un read-lock sobre A deberá tener un lock sobre la mayoría de copias de A. Para obtener un write-lock, se deberá tener lock sobre la mayoría de copias de A. El write requiere n+1 mensajes de control (n+1/2 para pedirlo n+1/2 para liberarlo) y n mensajes de datos. El número de mensajes de control de un read será igual que el write, la diferencia será que sólo necesita un mensaje de dato. Funciona bien en entornos donde predominan las escrituras. Hay menor riesgo de deadlock, dado que sólo un sitio puede tener mayoría.

K de N: Si hay N copias del dato, dado un valor K tal que N/2<K<=N. Para obtener un write-lock una transacción deberá tener lock sobre k copias de A. Para obtener un read-lock se deberá tener lock sobre N-K+1 copias de A. No es posible que existan read-lock y wirte-lock simultáneamente (ni dos write). A medida que k se incrementa, el protocolo se desempeña mejor en situaciones donde se realizan lecturas más frecuentemente. A medida que K se decrementa, funciona mejor si las escrituras son frecuentes.

Propuestas híbridas: sitio primario, token primario.

Sitio Primario: La responsabilidad del manejo sobre los bloqueos de un ítem A se

deja bajo al responsabilidad de un sitio en particular, sin importar cuantas copias del dato existan. Se identifica un sitio primario para cada ítem de dato. Si el sitio primario de A no está en el mismo nodo donde se ejecuta la transacción se deberá mandar un mensaje al sitio primario y esperar por una respuesta. Se distribuye así el trabajo, haciendo el sistema más robusto ante fallos.

Token Primario: Asume la existencia de read tokens y write tokenss. Para un ítem de dato A sólo existe un write token, pueden existir cualquier número de read-tokens. Si una transacción en un sitio desea obtener un write-lock para a deberá obtener el write token de A, si lo tiene genial; si no lo tienen envia n mensajes a todos los sitios pidiéndolo. Los sitios contestarán que no lo tienen, están por liberarlo, o lo tiene y no lo liberará. Si sucede uno de los dos primeros (nadie lo tiene o están por liberarlo) mandará n mensajes diciendo que aceptó el token y deberán destruir los suyos. Si alguno contesta que no lo liberará falla el lock y deberá mandar mensaje a los que contestaron que no lo tenían o lo iban a liberar para cancelar la reserva.

En un sistema distribuido se distingue entre transacción global y local. Se requiere asegurar atomicidad a nivel global así una transacción distribuida cometerá si y sólo si todas las subtransacciones que las forman están por cometer. Debe además asegurar serializabilidad a nivel global, es decir le ejecución distribuida debe ser equivalente a una ejecución en serie.

Para utilizar el PB2F hay que hacer algunas modificaciones: en transacciones distribuidas no sólo debemos controlar que realicen los bloqueos en dos fases localmente sino a nivel de transacción global. Además para asegurar atomicidad se dará que o todas las subtransacciones cometen, por lo que la transacción comete, o alguna subtransacción falla en cometer y la transacción aborta.

Protocolo de Compromiso Distribuidos

Sea T una transacción que fue iniciada en un sitio S y dividida en subtransacciones ejecutándose en sitios diferentes, incluyendo S. La subtransacción en S es el coordinador, las otras son participantes. Cada participante manda un mensaje al coordinador con su decisión de cometer o abortar, luego el coordinador toma la decisión final en función de las votaciones. Si recibe todos ready T, manda un mensaje a todos los participantes indicando un commit. Caso contrario, manda un mensaje de abort T. Cuando los participantes reciben el mensaje de cometer realizan el compromiso, escriban y vuelcan los registros de bitácora al almacenamiento no volátil y liberan los bloqueos.

El PCD podría generar estados de bloqueo donde una subtransacción no puede ni cometer ni abortar, si un participante no recibe ningún mensaje con la decisión, deberá mantener sus locks en espera de éste.

Protocolo de Compromiso de 2 Fases

Sea T una transacción que fue iniciada en un sitio S y dividida en subtransacciones ejecutándose en sitios diferentes, y sea C el coordinador de transacciones de esa localidad. Cuando T termina de ejecutarse C comienza el protocolo de compromiso de dos fases: primero recopila información de los diferentes sitios para saber si las subtransacciones pueden cometer; luego, en función de las respuestas determina si la transacción global comete o no.

En la primer fase C carga el registro <prepare T> a la bitácora y lo graba en memoria estable, luego envía un mensaje <prepare T> a los sitios correspondientes, finalmente cada gestor de transacciones que recibe el mensaje determina si puede

cometer o no su fracción de T, agregando su voto a la bitácora y enviando el mensaje a C.

En la fase 2 C recibe respuesta de todos los sitios o vence su timeout. Si recibe todos ready, agrega el registro a bitácora y manda los mensajes a los sitios. En otro caso se graba abort T en bitácora y se le informa a los sitios.

De fallar un participante y requerir recuperación, bastará con ver la bitácora para determinar el estado en que quedó (commit o abort). De sólo haber en bitácora un reay T, el sitio podrá mandar un mensaje al coordinador (help-me-T) para decidir sobre T, si C le responde con commit, habrá que hacer redo; si responde con abort, se hará undo; si no contesta se comunica con otros sitios.

Si llegara a fallar el coordinador en el medio de las dos fases, puede darse el caso que los participantes se comuniquen entre ellos y tomen una decisión entre ellos; o los participantes deberán esperar a que se recupere el coordinador. El coordinador revisará la bitácora para determinar que hacer.

Protocolo de Compromiso de Tres Fases

La primera fase es la de votación, idéntica al PC2F. C graba <prepare T> en bitácora y manda los mensajes, los participantes votan, agregando su vota a bitácora y mandando el mensaje al coordinador.

En la segunda fase, de precompromiso, si C recibe un mensaje para abortar o falla en llegar un mensaje, decide abortar y manda el mensaje a los participantes. Si recibe el ready T de todos los participantes toma la decisión de precommit y envía el mensaje a los participantes. Cuando los participantes reciben el mensaje del coordinador, guardan en bitácora dicho mensaje y retornan un mensaje de <acknowledge T> al coordinador.

La tercer fase se ejecuta sólo si se da el commit. Una vez recibidos los <acknowledge T> graba el commit en la bitácora y manda el mensaje a los participantes. El rol de la fase 3 se justifica en caso de fallos.

El PC3F exige que no ocurran particiones de la red, a lo sumo k sitios participantes puedan fallar mientras se ejecuta el protocolo. En cualquier momento debe haber k+1 sitios funcionando correctamente.

Sistemas de Tiempo Real

Sistemas en los que la correctitud está relacionada con la lógica y la implementación de los programas así como también el tiempo en que entregan su resultado Si las restricciones de tiempo no se respetan se dice que el sistema ha fallado. Son sistemas que interactúan con personas y ambientes, y que dependiendo del sistema, una respuesta fuera de tiempo puede ser catastrófica. Debe tener un buen manejo de interrupciones, una asignación de prioridad y un control del entorno. Según las restricciones temporales pueden ser:

- Con deadline estrictos: debe cumplir con las restricciones pues es crítico.
- Con deadline firme: la tarea no tiene valor si se completa luego del deadline.
- Con deadline soft: las restricciones de tiempo no son críticas, el resultado no pierde valor.

Las aplicaciones de tiempo real son cada vez más complejas y requieren acceso a mayor cantidad de datos. Es por eso que se considera combinarlos con las DBMS. Los sistemas de base de datos de tiempo real deben satisfacer las restricciones temporales que caracterizan a los RTS, y las restricciones de consistencia de las DBMS. Con los RTDBS se asume que el acceso a los datos es aleatorio lo que hace complicado el proceso de planificación y difícil garantizar todas las restricciones de tiempo. Es por esto que se

busca minimizar las violaciones.

Algoritmos para Planificación Concurrentemente

En los algoritmos de planificación para sistemas de bases de datos con restricciones temporales se identifican 4 componentes:

- una política para administrar sobrecarga.
- Una política para asignar prioridades a las tareas.
- Un mecanismo de control de concurrencia: una política para resolver conflictos entre dos o más transacciones que necesitan bloquear el mismo dato.
- Una política para planificar requerimientos de I/O.

Cada transacción T tiene asociado un tiempo de arribo R, un deadline D y un tiempo estimado E, que es una aproximación al pero tiempo de duración de T. Estos parámetros se conocen cuando T arriba.

En un RTDBS se necesita asignar prioridades a las transacciones, éstas deberán ser tenidas en cuenta al momento de atender los pedidos y armar la planificaciones.

First Come First Served: se sirve primero al que llega primero. No es aplicable cuando las transacciones no tienen todas el mismo peso.

Deadline Más Próximo: da mayor prioridad a la transacción que tiene el deadline más cercano. La desventaja es que puede asignar mayor prioridad a una transacción vencida o a punto de vencer.

Slack Time (Resto Menor): slack time será **S=D-(t+E-p)** donde D es el deadline, t es el tiempo actual, E es el estimado de ejecución y p es el tiempo que lleva ejecutando. S estima cuanto puede demorarse T y aún así cumplir su deadline. Se puede evaluar el slack time estáticamente al momento de arribo de la transacción, o dinámicamente cada vez que se necesita conocer la prioridad de una transacción.

Ciertas modificaciones al mecanismo de **control de concurrencia** usando bloqueos basados en el modelos de control de concurrencia de dos fases permiten adaptar la forma de generar planificaciones con restricciones temporales.

PB2F no considera la prioridad de las transacciones lo que puede llevar a que una transacción de mayor prioridad sea bloqueada por una de menor prioridad. Dadas las transacciones TH y TL con prioridades TH>TL. Si ambas acceden al mismo dato, pero TL obtiene el bloqueo, si TH pide el bloqueo, se verá bloqueado por TL quién tiene menor prioridad.

Mayor Prioridad:el esquema 2PL mayor prioridad resuelve todos los conflictos por acceso a los recursos a favor de la transacción de mayor prioridad con apropiación. La transacción TH en ese caso se apropiaría del dato y TL sería retrocedida. Mayor Prioridad es similar al esquema Wound-Wait pero difiere en la forma de asignar prioridades. En el wound-wait las prioridades estan basadas en el tiempo de arribo de las transacciones en vez de las restricciones temporales.

Esperar y Promover: resuelve los accesos conflictivos haciendo esperar a la transacción que hace el pedido. Además incluye un mecanismo de herencia de prioridades, de manera tal que si la prioridad de la transacción que tiene el lock es menor que la que lo pide, la que retiene el pedido heredará la prioridad del que pide hasta que la transacción termine. Esto acelera la transacción de menor prioridad y evita que transacciones de prioridad intermedia frenen a la de mayor prioridad.

Herencia de Prioridad Condicional: abortar transacciones próximas a terminar

resulta un desperdicio del tiempo de CPU, sin embargo, no abortar transacciones que bloquean a otras de mayor prioridad puede ocasionar que dichas transacciones no cumplan con sus restricciones temporales. El esquema herencia de prioridad condicional mezcla los dos métodos anteriores. La herencia de prioridades sólo es usada en caso de que la transacción bloqueante este a punto de terminar, caso contrario se aborta. Específicamente, éste método se usa si lo que queda por ejecutar de la transacción es menor que un límite h. La dificultad está en definir h.

Una idea similar se encuentra en el Reinicio Condicional, aquí la decisión de abortar o no a TL se da calculando el slack time de TH. Si TH cuenta con tiempo suficiente para que TL termine su ejecución se procede a la herencia de prioridades, caso contrario, se aborta.

Sistema: conjunto de ítems interrelacionados que de forma ordenada constituyen un todo. Procedimiento organizado y establecido.
Sistema Automatizado: sistemas hechos por el hombre controlados por una o más computadoras. En general se compone de Hw, Sw, personas, datos, procedimientos, documentación.
Batch: recolectan datos por un período de tiempo, no interactuan con usuarios, procesan varias transacciones juntas, tienen acceso secuencial a la mayoría de la información.
On-Line: la transacción se registra cuando sucede, procede de una por vez, interactua con el usuario, requiere acceso rápido a los datos, se accede de forma aleatoria a una porción de los datos, las transacciones son sencillas.
De Tiempo Real: controlan un ambiente recibiendo datos, procesándolos y devolviéndolos con suficiente rapidez como para influir dicho ambiente.
De Soporte de decisión: no toman decisiones por sí sólos sino que colaboran con la toma de decisión, no poseen salidas programadas, pueden presentar la información de varias maneras.
Basados en Conocimientos: sistemas expertos, imitan un comportamiento, utilizan técnicas de IA.

Sistema de Software: es una colección de componentes de software interrelacionados que trabajan conjuntamente para cumplir algún objetivo. Aún los sistemas de Sw más simples se componen de varios componentes, el funcionamiento exitoso de cada componente depende del funcionamiento correcto de los otros componentes. En general, los sistemas son jerárquicos e incluyen a otros sistemas (subsistemas).

Un sistema automatizado forma parte de un sistema mayor. Los sistemas automatizados reemplazan a sistemas que existían previamente. Aunque los distintos tipos de sistemas parecen diferentes, existen principios, teorías y filosofías que son compartidos por todos.

La producción de Sw evolucionó con el tiempo, primero consistí en ubicar instrucciones juntas para que la computadora haga algo útil. Luego con los lenguajes de alto de nivel y las computadoras más accesibles, se distinguió el rol entre programador y usuario y empezaron a surgir proyectos de Sw de mayor escala (SO). La Ingeniería de Sw trata el Sw como parte de un sistema más complejo.

Ingeniería de Software: es un área de las ciencias de la computación que estudia

la construcción de sistemas de Sw tan grandes y complejos que requieren de un grupo de ingenieros. La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del Sw, esto es aplicación de ingeniería de Sw. Es esencialmente una actividad en equipo: un ingeniero de software desarrollará un componente que se combinará con otros desarrollados por otros ingenieros. Existen versiones del producto. El producto perdurará en el tiempo y está sujeto a cambios. Requiere de un trabajo disciplinado.

Los Principios de Ingeniería de Sw son enunciados generales y abstractos que describen las propiedades deseables de los procesos y productos de Sw. Para aplicar los principios se requiere métodos, guías generales que gobiernan la ejecución de alguna actividad, y técnicas, guías más mecánicas que los métodos. Las metodologías proveen una aproximación segura para resolver un problema, preseleccionando los métodos y técnicas a ser empleadas. Los Principios de la Ingeniería de Software son: Rigurosidad y Formalismo; Separación de los Intereses; Modularización; Abstracción; Anticipo al cambio; Generalidad; Incrementabilidad.

Proceso de desarrollo: define el marco de trabajo para un conjunto de tareas calves en la producción de Sw. Generalmene, en cualquier proceso de ingeniería de Sw se puede dividir en tres fases genéricas: definición, que se espera del producto; desarrollo, cómo se va a hacer; y mantenimiento.

Producto de IS: es un sistema de software que se distribuye al cliente junto con su documentación. La IS apunta a producir Sw como una actividad de ingeniería. Los productos de Sw pueden ser a medida, desarrollados para un cliente en particular bajo contrato, o genéricos, desarrollados para ser vendidos a un mercado abierto.

A diferencia de otros productos de ingeniería el Sw es lógico y no físico, se desarrolla, no se fabrica. El Sw no se estropea, pero se deteriora.

Se busca desarrollar productos de IS de alta calidad, existen diferentes enfoques para la calidad. La calidad puede ser interna o externa. La interna tiene interés para los desarrolladores, la externa para el usuario.

Externa

- Correctitud: si se comporta según lo especificado. (Producto)
- Confiabilidad: si es correcto y robusto, si el usuario puede depender de él. (Producto y Proceso)
- Robustez: si se comporta de forma razonable ante circunstancias no especificadas. (Producto y Proceso)
- Eficiencia: si usa los recursos de forma económica. (Producto)
- User freindly: si el usuario lo encuentra adecuado para trabajar. (Producto y Proceso)
- Interopperabilidad: la habilidad de un sistema de coexistir con otros. (Producto)

Interna

- Verificabilidad: si se pueden comprobar sus propiedades. (Producto y Proceso)
- Mantenibilidad: la capacidad de introducir modificaciones. (Producto y Proceso)
- Reparabilidad: capacidad de corregir errores del Sw en tiempo de trabajo limitado. (Producto)
- Evolutividad: capacidad de adaptarlo a nuevos requerimientos. (Producto y Proceso)
- Reusabilidad: capacidad de reutilizarlo haciendo cambios menores. (Producto y Proceso)
- Comprensibilidad: mide el diseño desde el punto de vista de su compresión.

(Producto, puede ser externa también)

- Productividad: mide la eficiencia del proceso. (Proceso)
- Puntualidad: la habilidad de entregar el producto a tiempo. (Proceso)
- Visibilidad: si sus etapas están claramente documentadas. (Proceso)

El proceso define un marco de trabajo para un conjunto de tares calves en la producción del Sw. El ciclo de vida de un sistema comprende desde que la concepción de la idea de un desarrollo de Sw hasta que es implementado, entregado y aún después que deje de usarse.

Code-Fix: primera aproximación de desarrollo, no es un modelo de proceso. El desarrollo de Sw era una tarea unipersonal, el problema sencillo y bien entendido, los lenguajes de bajo nivel. Consistía en escribir el código y luego corregirlo.

Hitos: se separan los roles de usuario/programados. Necesidad de aplicaciones más complejas. El desarrollo se convierte en una actividad grupal.

1. Estudio de Factibilidad: se evalúan costos y beneficios.
2. Análisis y especificación de los requerimientos: identificar las calidades de la aplicación. Identificar que, como. La clave es separar, abstraer y particionar.
3. Definición de la arquitectura, diseño detallado del sistema.
4. Codificación y testeo de Módulos.
5. Integración y testeo de sistema.
6. Distribución, entrega, mantenimiento.

Modelo de Cascada: las tareas de desarrollo se organizan en cascada, una después de la otra. La salida de una etapa es la entrada de la otra. Se usó para las primeras aplicaciones es crítica y compleja. Es un modelo secuencial, cada fase debe terminar antes de pasar a la siguiente. La definición de las actividades depende del tipo de proyecto y la relación entre el equipo de desarrollo y el cliente. Cada actividad produce un documento de salida que el el ingreso para la siguiente actividad. Modelo Conducido por la Documentación. Introdujo la disciplina de proceso, pospone la implementación hasta que los objetivos están claros. Impone puntos de control claros.

Es muy rígido, retrasa la detección de problemas críticos, difícil de estimar los recursos, el usuario sólo interviene al inicio y al fin, no brinda anticipación a cambios, los costos se transladan al mantenimiento, no permite implementaciones parciales.

Paradigma Evolutivo: dado que las fallas de la primera versión de una aplicación conducen a la necesidad de rehacerla, plantea hacerlo dos veces. El problema esta entre la definición de requerimientos y la distribución de la aplicación la estrategia es entregar algo, y medir el valor agregado. Luego, ajustar el diseño y objetivos.

Primero elaboro un prototipo para descartar, a modo de validar los requerimientos y resolver aspectos críticos del diseño. Los objetivos son una investigación mejor de los requerimientos y diseño y reducir el riesgo que la aplicación no se ajuste a las necesidades del cliente.

Se hace una recolección de los requerimientos del sistema, se definen los objetivos globales. El diseño es rápido, centrado en los aspectos de Sw que son visibles al usuario.

Las desventajas son que el cliente confunde el prototipo con el sistema real, se toman decisiones rápidas difíciles de revertir, el prototipo para descartar nunca se descarta.

Modelo Iterativo Incremental: consiste en incrementos expandidos de un producto de software operativo, conducidos por la evolución determinada según la experiencia operativa. Los incrementos se distribuyen a medida que se completan.

Las ventajas son que el usuario ve algo rápidamente, se piensa en su calidad desde el principio, los ciclos van mejorando con las experiencias de los anteriores, los trabajadores del equipo de desarrollos se especializan en ciertas actividades.

Es difícil mantener el foco, los incrementos entran en mantenimiento mientras se están desarrollando nuevos. Es difícil seguir procesos puramente iterativos incrementales.

Modelo Espiral: combina la actividad de desarrollo con la gestión de riesgo, para minimizar y controlar el riesgo. El riesgo son circunstancias potencialmente adversas que pueden perjudicar el proceso de desarrollo y la calidad de los productos. Es cíclico, es un meta-modelo. La estrategia es comenzar por los desarrollos de más riesgo.

En una primera etapa se identifican los objetivos de la porción del producto bajo consideración, en términos de calidades a lograr. En la segunda etapa se evalúan las alternativas y se identifican las áreas de riesgo potencial. En la tercera etapa se desarrolla y verifica el próximo nivel del producto. En la cuarta, se revisan los resultados para planificar la próxima vuelta espiral.

Constituye un enfoque realista del desarrollo de sistemas de software de grane escala. Desarrollador y cliente comprenden y manejan mejor los riesgos. Mantiene el enfoque sistemático de los pasos sugeridos por el ciclo de vida cascada, pero incorpora el marco de trabajo iterativo. Considera los riesgos técnicos en todas las etapas.

Requiere de habilidades para la evaluación del riesgo, si un riesgo no se descubre pierde eficiencia. No existe demasiada experiencia en sus uso.

Modelo Transformacional: basado en especificaciones formales, el desarrollo de Sw como una secuencia de pasos que gradualmente transforma una especificación a un implementación. Se debe transformar requerimientos informales en especificaciones formales, pretende reducir errores automatizando pasos del desarrollo.

Busca reusar y construir componentes reusables. Los cambios se hacen en la especificación, todo queda documentación garantiza resultados correctos.

Es caro y lleva tiempo. Se requiere de estudios detallados y personal capacitado. Difícil de utilizar el método como medio de comunicación con el cliente.

Desarrollo Rápido de Aplicaciones: apunta a proyectos que se terminan en 60-90 días. Utiliza un enfoque de construcción basado en componentes. Requiere muchos recursos humanos para crear varios equipos DRA, tanto desarrolladores como clientes deben estar comprometidos