Disclaimer: Este apunte no es autocontenido y fue pensado como un repaso de los conceptos, no para aprenderlos de aquí directamente.

1. Introducción (capítulo 1)

DBMS:
- Persistir datos
- Acceso eficiente
- Otros servicios
  - Soporte de uno o más modelos de datos (relacional: tupla $\in$ relación)
  - Soporte de lenguajes de consulta de alto nivel
  - Transacciones/concurrencia
  - Control de acceso/vistas
  - Recuperacion ante fallas

División en capas (proveer independencia del soporte y niveles de abstracción)
- Física (implementación del motor)
- Conceptual (diseño de la base)
- Vista (uso de la base)

Subesquema: Vista

Lenguajes:
- De definición de datos (define relaciones)
- De manipulación de datos (define tuplas)

DML: Data model language, lenguaje de la BD $\neq$ Host language (lenguaje del resto del sistema) es porque DML no es Turing-completo, pero esa falencia permite buenas optimizaciones.

Integración de DML con Host language
- DBMS de objetos (identidad de objetos, tuplas en rel pueden estar repetidas)
- Logica (Prolog, relaciones definidas por extensión)

KBMS: (Knowledge) DBMS + lenguaje *declarativo* y "total"

2. Organización Física (capítulo 6)

2 niveles: Memoria y de acceso secundario. Costo = cantidad de transferencias de bloques de memoria secundaria a RAM

Pointers: A registros (la implementación exacta depende del soporte)
- Dado un puntero, puedo obtener el registro apuntado (*)
- Dado un registro, puedo obtener el valor de un puntero a el (&)
- 2 registros $\neq$ tienen $\neq$ puntero.

Registro "pinneado": So tiene punteros desconocidos $\Rightarrow$ limita los movimientos que se pueden hacer porque estos invalidan los punteros

Cuidado al borra $\Rightarrow$ Marcar como borrado para no invalidar punteros y que apunten a fruta.

Contenido registro: Datos [+ Bit borrado] [+ Bit usado/libre] [+ info tipos] [+ longitud] [+ basura (para alinear)]

Longitud variable:
- Longitud en cada campo (-espacio)
- Puntero a cada ppio de campo (-tiempo de acceso)
  - Se usa para registros dentro de bloques

Registro semi-"pinneado": Se puede mover dentro del bloque, un nivel de indireccion de los punteros de afuera (tambien borrar sino lo borras del diccionario dentro del bloque, que debe ser mas grande que la cantidad máxima de registros que entran para que tenga sentido)

Notación: R es la cantidad de registros por bloque y n la cantidad de tuplas de la relación.

Heap-file:

Find: $\lceil n/R \rceil$ peor caso, $\lceil n/2R \rceil$ caso promedio
Insert: 2 (1 lectura y 1 escritura al final)

Mod/delete: Lookup + 1 (para escribir)

Hashed files:

$h :: Key \to [0..B-1]$ (B número de buckets)

hay B heap files, el costo es el costo del heap file con n =cant en el bucket

en general las operaciones son $1/B$ mas rápidas (+o-, según cuan bueno sea h) obvio que $B > n$ es al pedo

Si el directorio de buckets no está en memoria, a todas las operaciones hay que agregarle 1 para acceder a él.

Indexed files:

mantener el archivo ordenado por clave. Indice primer valor-¿bloque (solo se escanean el indice y el bloque posta).

Binary search: Mejor aun, reduce la parte de escaneo del indice

Interpolation search: Como BS pero en lugar de (a+b)/2 elegir pivot basado en estadísticas.

Si los regs son pinned hay que suar el orden "inicial" como un hash con buckets, porque no se puede insertar ordenado. Se pueden enlazar punteros para poder recorrer ordenado facilmente.

B-Trees:

Árbol k-ario de búsqueda. En cada nodo hay $2d-1 \geq 3$ números ordenados y $2d$ punteros según entre cual y cual (o antes del primero o despues del último) está el que se busca.

Invariante: Cada nodo está al menos a medio llenar (salvo el root). Si se pasa parto en 2 y divido el nodo, insertando arriba. El árbol crece en altura solo cuando se divide el root (se crea un nuevo root). Las hojas con los datos tienen $2e-1$ cosas cada una.

Lookup: $1 + \log_d(n/e)$. Otras operaciones $2 + \log_d(n/e)$ promedio. Peor caso patológico podría ser hasta $3(1 + \log_d(n/e))$ aproximadamente.

Índice denso: Contiene puntero a registro, no a bloque ⇒ ahorra espacio y permita que los regs sean pinnead (manteniendo la entrada del índice unpinned) pero es un poco menos eficiente.

Salteado: 6.7

Índice secundario: (no sobre la clave) map<value,ref>

Ref puede ser un puntero (pinnea) o una clave (no pinnea, pero es menos eficiente).

Pueden ser densos, warning, si un bloque tiene A-B y otro B-C las primeras que empiezan con B pueden estar en el 1er bloque y hay que probar (si el valor es clave como antes, esto no pasa porque no hay repetidos)

Salteado: 6.9, 6.10, 6.11

2.1. Queries por rango o parciales

Arbol B+ y sorted file sirven para rango, hash no.

Problema de siempre: múltiples resultados (se puede hacer la intersección con punteros y evitar mover todo un reg grande que después no se usa)

Hash particionado: Hashear cada campo por separado y después concatenar (permite buscar por subclaves)

Salteado: 6.14

3. Transacciones / concurrencia (capítulo 9)

Transacción: Secuencia de escrituras y lecturas entre espacio privado y la base de datos

Atomicidad: Se hace del todo o no se hace y nada se hace en el medio (aparenta comportarse así)

Seralización

- Tiempo compartido ⇒ una transacción se puede interleavear con otras
- Aborts (no debe verse ningún efecto en la base o en otra transacción)

Granularidad: Tamaño del átomo (ítem) que se lockea (campo, reg, bloque)

- La óptima es cuando cada transacción lockea unos pocos ítems

Locks: Permiso de acceso. Se mantienen en una tabla <ítem,lock type, trans>.

Schedule: Secuencia de pasos de varias transacciones posiblemente interleaveadas

Legal: No se comparten nunca locks que no vale compartir

Livelock: Starvation (se puede solucionar con un scheduler como FIFO)

Deadlock: T1 Lock A, T2 Lock B, T1 Lock B, T2 Lock A

- Soluciones
  - Pedir todos los locks juntos (se dan todos o ninguno)
  - Pedir todos los locks en un órden arbitrario
  - Permitir deadlock y detectarlo, cuando se detecta abortar alguna (solo está bueno cuando la probabilidad de deadlock es baja)

Schedule serial: Cada transacción ejectua toda seguida

Resultado razonable $\Leftrightarrow$ equivale a un schedule serial cualquiera

- Resultado de un item es igual en 2 schedules distintos $\Leftrightarrow$ se le hacen las mismas operaciones en el mismo órden (ignoramos propiedades algebráicas de las operaciones porque sería muy dificil detectar ese tipo de igualdad). Esto solo produce falsos negativos, pero no falsos positivos que sería mucho mas grave (causa inconsistencia).

Scheduler: Árbitro de transacciones (prevenir y controlar conflictos)

- Hace que las transacciones esperen o aborten cuando piden locks no disponibles.

Protocolo: Restricción en la secuencia de pasos (instrucciones) de una transacción. Ej: Pedir locks en órden

3.1. Modelo simple de locking: lock/unlock

Semántica:
A cada par correspondiente de lock A - unlock A asociar una función distinta f_i. Sea A_0 el valor incial de A. En cada unlock A se reemplaza la formula en ese momento para A por la f_i que corresponde a ese unlock, aplicada a todos los items que tengan sus locks antes que el unlock de A (esto último incluye a A).

2 scheuldes son equivalentes $\Leftrightarrow$ la fórmula en cada ítem es la misma

Test de seralizabilidad: Buscar un topological sort en el grafo de serializacion $G = (V, E)$ donde $V =$ conj de trans y $E = \{T_i \rightarrow T_j | (\exists A)$el primer lock de A despues de que T_i haga unlock A es de $T_j\}$. Si hay ciclos (i.e., no hay topological sort) el conjunto no es serializable.

Demostración de correctitud:

$\Rightarrow$) Supongamos que hay un ciclo y T_j es la primer transacción en una serialización válida que pertenece a un ciclo en el grafo. Sea T_i la transacción anterior a T_j en el ciclo ($T_i \rightarrow T_j \in E$). Sean f_i la función asociada al unlock A de T_i que produjo el eje y f_j la asociada al lock A en T_j. En la semántica de la serialización el paso por el unlock A de T_j correspondiente a ese lock aplicara f_j a una expresión y luego el unlock A de T_i (viene despues por hipótesis) le aplicará f_i a una expresión que contiene f_j, en cambio en el schedule original el unlock A de T_i le aplicará f_i a una expresión que no contiene f_j (ya que el unlock A de T_j que corresponde a f_j está luego del lock A en T_i, que a su vez esta luego del unlock A en T_i porque existe el eje $T_i \rightarrow T_j$ en el grafo). Como cada función aparece a lo sumo una vez en la expresión de cada ítem, no pueden ser iguales ambas expresiones ya que una tiene f_j dentro de f_i y la otra no.

$\Leftarrow$) G es acíclico, sea la profundidad de T la longitud del máximo camino que termina en T. T_i solo lee cosas escritas por transacciones de profundidad menor (por construcción del grafo) asi que haciendo inducción en la profundidad d, toda transacción escribe el mismo valor para todo ítem en el schedule original y en la serialización ya que los valores obtenidos son originales o fueron escritos por transacciones de profunidad menor.

Protocolo en 2 fases: Toda transacción tiene todos sus locks antes que todos sus unlocks

- Teorema: Cualquier schedule legal que cumple 2-phase protocol es serializable. Demo: Sino, el grafo de serializacion tiene un ciclo, como los ejes son unlock-¿lock, tener un ciclo implica que hay un unlock antes que un lock en toda transacción perteneciente al ciclo.
- Optimalidad: Si T no es 2-phase $\exists T'$ tq {T,T'} tienen un schedule legal no serializable. T =Lock A,Unlock A,Lock B,Unlock B. T' =Lock A, Lock B, Unlock A, Unlock B y metiendo T' despues del Unlock de A (hay eje entre ambas para los dos lados).

3.2. Modelo con read-lock y write-lock

Semántica: Igual que el anterior, pero solo cambiar el valor de A por $f_i(...)$ si el unlock corresponde a un write-lock.

2 schedules son iguales $Leftrightarrow$ todo ítem tiene igual valor al final y cada ítem tiene igual valor en ambos schedules al momento de sus read-lock.

Test de serializabilidad: Misma idea, pero solo pongo eje entre unlocks y locks que no sean "read-read"

- Demostración igual al anterior, con un par más de casos.

2-phase también provoca legal $\Rightarrow$ seralizable, y también es óptimo.

3.3. Modelo mas general

Matriz de compatibilidad de locks (cuáles parejas se pueden dar al mismo tiempo y cuáles no).

Grafo para test: Hay que poner todos los ejes cuando hay unlock y luego un lock incompatible (no se puede optimizar como antes con "solo el próximo" porque no hay necesariamente transitividad como antes -salvo que la matriz la implique-)

Salteado: 9.6,9.7

Fallas:

- Por abort (culpa de la transacción o forzado por scheduler
- Por fallas del sistema

Agregamos commit como operación.

Problema: Leer datos sucios (escritos por alguién que luego aborta), provoca rollback en cascada.

2 fases estricto: Se escribe y se liberan todos los locks luego (durante) el commit

- Previene rollback en cascada (aparte incluye el 2-fases y sus beneficios)

Protocolos agresivos: Hacer todo los posible **Protocolos conservativos:** Evitar conflictos

La elección de protocolo y scheduler afecta:

- Cantidad de ciclos de reloj que consume la actividad del scheduler
- Ciclos desperdiciaciones por aborts
- Ciclos para restaurar la DB luego de aborts

Conservador: Pedir todos los locks al principio (no deadlock) + FIFO estricto (no livelock). Este último podría relajarse a uno que permita schedulear alguien que tiene todos sus locks listos y ninguno mas adelante en la cola necesita uno de esos locks.

- Problemas: Delay innecesario (una transacción puede estar obligada a no empezar a hacer su proceso solo porque no tiene disponible un lock que necesita solo al final), obliga a pedir locks que quizas ni usas, los locks se mantienen usados por mucho tiempo al pepe lo que aumenta el bloqueo de la cola.

Agresivo: Pedir cada lock lo + tarde posible (si se va a leer y luego a escribir pedir primero de lectura y luego upgradear) y liberar apenas se termina de usar.

- Problemas: puede tener deadlock (el upgrade genera un nuevo tipo de deadlock cuando 2 tienen lectura del mismo y los 2 quieren upgrade a escritura).

3.4. Restauración frente a fallos

Logging: Guardar un conjunto de tuplas <trans,ítem,oldValue,newValue>

- De los valores se pueden guardar ambos o solo 1 (según el tipo de log)

Es mejor escribir log todo el tiempo que la base todo el tiempo ya que tiene mucha mas localidad espacial (se escribe de corrido).

Redo log: Solo el newValue. Poner <T,start> al iniciar y <T,commit> o <T,abort> al finalizar. Los que se encuentran sin commit ni abort es porque se corto el proceso en el medio. Al recuperar, para cada T que tenga su commit rehacer las operaciones en orden de mas vieja a mas nueva. Al final agregar aborts a las uqe hayan quedado interrumpidas. No hace falta que la escritura sea inmediatamente a la base, ya que se puede volver a rehacer luego de ocurrir un nuevo error.

Checkpoint: Cuando hay 0 transacciones activas y todo resultado ha sido escrito en la base en disco, se puede hacer un checkpoint y las próximas recuperaciones solo se hacen desde ahí. De paso, permite descartar todo el log anterior a un checkpoint, ya que no se precisa volver a usarlo.

3.5. Control de concurrencia con timestamps

Es mas bien agresivo.

Serializable por timestamps: El órden del serial es el órden de los timestamps (se interpreta que cada transacción se hizo atómica e instantáneamente en el momento en que se inicia, momento en el que recibe su timestamp)

Serializable por timestamps $\neq$ serializable por locks (y ninguno $\subset$ en el otro)

Los timestamps pueden ser cualquier cosa creciente, pero se debe asegurar todos disjuntos (si hay varios CPUs se puede usar clock concatenado con identificador único de CPU).

Al igual que con los locks, se pueden distinguir tipos de acceso a un ítem. Cada ítem tiene un tiempo por cada tipo de acceso. Cada vez que se va a realizar uno se chequea que sea consistente con los timestamps de todos los tipos. En algunos casos (como write-write) se puede dejar todo consistente ignorando la escritura (si voy a escribir sobre algo que se supone fue escrito despues que mi propio timestamp, simplemente no hago nada).

Mantenimiento: Mantener el timestamp para todos los items de la base es muy costoso, pero como solo importan los timestamps $\geq$ el minimo actualmente activo, se puede mantener este último e ir descartando los timestamps que quedan mas viejos que este, manteniendolos en un diccionario de ítem en timestamp

Hay que hacer rollback en cascada cuando se usa éste método, usando log undo/redo (el log redo solo no sirve para rollback en cascada, porque lo que se precisa es deshacer las transacciones que leyeron sucio).

Salteado: 9.11 (ts-estricto, multiversion, restart)

4. Dependencias y normalización (capítulo 7)

Dependencias: Solo deja como legal un subconjunto de los posibles contenidos de una relación
- Solo dependen de la igualdad (o desigualdad) de valores, no aritmética

$X \to Y$ "X determina funcionalmente a Y"

X clave $\Rightarrow X \to Y$ $(\forall Y \subset R)$

Dependencias funcionales: Statement aerca de cualquier instancia r de R

$F| = X \to Y$ si r satisface $F \Rightarrow r$ satisface $X \to Y$

$F^+ = \{X \to Y | F| = X \to Y\}$

Clave: X clave $\Leftrightarrow X \to R \in F^+ \wedge \neg\exists Y, Y \subset X, Y \to R \in F^+$

Superclave: X superclave $\Leftrightarrow X \subset Y \wedge Y$ es clave

Axiomas de Armstrong:
- Reflexividad: $Y \subset X \Rightarrow X \to Y$
- Aumento: $X \to Y \Rightarrow XZ \to YZ$
- Transitividad: $X \to Y \wedge Y \to Z \Rightarrow X \to Z$

Demostración soundness:
Reflexividad y aumento son triviales, transitividad sale viendo que por absurdo si $\mu[X] = \nu[X] \wedge \mu[Z] \neq \nu[Z]$ entonces $\mu[Y] \neq \nu[Y]$ (por la segunda dependencia) violando la primera.

Otras reglas:
- Union: $X \to Y, X \to Y \ \Rightarrow X \to YZ$, aumento $X \to YX$, aumento $YX \to YZ$, trans
- Descomposicion: $X \to AY \Rightarrow X \to A$ refl $A \to A$, aumento con Y, trans
- Pseudotransitividad: $X \to Y, WY \to Z \Rightarrow WX \to Z$ aumento $XW \to WY$, trans

Demostración completeness:
Lema: $X \to Y$ se sigue de F usando Armstrong $\Leftrightarrow Y \subseteq X^+$ donde $X^+ = \{A | X \to A$ se sigue con

Armstrong de $F\}$. Sea $Y = A_1, ..., A_n$

$\Rightarrow$) Por descomposición. $X \to A_i$ se sigue de $F \Rightarrow A_i \in X^+$ (por def).

$\Leftarrow$) $\forall i) X \to A_i$ se sigue de F (por def de X^+) $\Rightarrow$por unión, $X \to Y$

Quiero ver que $F| = X \to Y \Rightarrow X \to Y$ se deriva por Armstrong de F.

Sea $X \to Y$ tal que no se deriva de F por Armstrong y sea r:

X^+	otros atributos
$1 \dots 1$	$1 \dots 1$
$1 \dots 1$	$0 \dots 0$

(*1) r cumple F: si $V \to W$ no cumple $V \subseteq X^+$ y $\exists A, A \in W, A \notin X^+$ (por definicion). Ahora, por refl $X \to V$ se sigue por Armstrong y $V \to W \in F$ asi que por transitividad $X \to W$ se sigue por Armstrong y por refl+trans $X \to A$ se sigue por Armstrong $\Rightarrow A \in X^+$ por definición, absurdo.

Si $X \to Y$ es satisfecho por r como $X \subseteq X^+ \Rightarrow Y \subseteq X^+$ (sino no cumple por la forma de r) $\Rightarrow$por lema que $X \to Y$ se deriva de F por Armstrong.

Con esto, podemos redefinir las clausuras de F o de X en cualquiera de ambos sentidos (ambos o para toda relación) y da igual.

4.1. Algoritmo para determinar X^+