Bases de Datos

Bibliografía:

- Diapositivas de clase, Universidad de Murcia.
- Wikipedia, the Free Encyclopedia. *Augmented Backus-Naur Form* (https://en.wikipedia.org/wiki/ABNF).
- Digital Equipment Corporation (30 de julio de 1992), *ISO/IEC 9075:1992, Database Language SQL* (http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt).
- SQLite. *Query Language Understood by SQLite* (https://sqlite.org/lang.html).

Capítulo 1

Introducción

Un **dato** es un hecho conocido que puede registrarse. Una **base de datos** es un conjunto coherente de datos almacenados con significado inherente sobre un **dominio** o **universo de discurso** del mundo real, diseñada, creada y cargada con datos para conseguir ciertos objetivos para ciertos usuarios.

Un **modelo de datos** es una herramienta formal para comprender y representar el mundo real para conseguir una visión abstracta de un dominio, mediante tipos de datos, relaciones entre ellos y restricciones que deben cumplir, e incluye operaciones básicas para especificar consultas y modificar tanto los datos como su estructura. Tipos de modelo:

1. **Conceptuales** o **de alto nivel**: Describe la realidad como entidades que se relacionan entre sí. **Modelo entidad relación** (**MER** o **ERM**, *Entity-Relationship Model*), modelos **orientados a objetos** como **UML** (*Unified Modeling Language*).

2. **Lógicos** o **de representación**: Se ocultan detalles de implementación para que los conceptos sean entendibles por usuarios finales, pero son directamente implementables. Son los más usados. Distinguimos modelos lógicos **basados en registros**, como el **relacional**, de **red** o **jerárquico**, y **orientados a objetos**, más próximos a los conceptuales.

3. **Físicos** o **de bajo nivel**: Dirigidos a especialistas, describen la estructura física, incluyendo el formato y ordenamiento de los registros, los tamaños de página y de bloque y los caminos o estructuras de acceso a datos, como índices.

Un **sistema de gestión de bases de datos** (SGBD) es un software para definir el **esquema** de una base de datos según un modelo; crear la base de datos en un almacenamiento según el esquema; manipularla consultando, introduciendo, modificando o eliminando datos para reflejar cambios en el dominio y generar informes, y controlar el acceso a la base de datos proporcionando seguridad, integridad, control de concurrencia y recuperación tras fallos, y ofreciendo un catálogo de **metadatos**, descripciones de los datos.

Un **sistema de bases de datos** consta de una base de datos, un sistema de gestión y software de aplicación para usar la base de datos. El **estado** de una base de datos es el conjunto de datos, **instancias** de los elementos del esquema, que contiene en un instante concreto.

El diseño de una base de datos se hace para satisfacer los requisitos de contenido de usuarios y aplicaciones, estructurando los datos de forma natural, fácil de entender y flexible para

poder modificar el esquema posteriormente, y para soportar los requisitos de procesamiento y rendimiento. Pasos:

1. Recopilación de requisitos de usuario.
2. Diseño conceptual: Análisis de los requisitos y **conceptualización** o diseño del esquema conceptual, que describe los requisitos de forma independiente de usuarios, aplicaciones, SGBD específico o rendimiento.
3. Elección del SGBD.
4. Diseño lógico.
5. Diseño físico: Obtención de un **esquema interno** para conseguir una implementación lo más eficiente posible del esquema lógico en el SGBD elegido. Las decisiones en esta fase pueden afectar al esquema lógico, aunque esto no es conveniente.
6. Implementación.

Para el diseño usamos herramientas o recursos como diagramas, grafos, teorías, modelos, lenguajes de datos (como SQL), documentación, adquisición de requisitos, diccionarios de datos, etc. Un **método** es un conjunto de herramientas para facilitar la representación de los datos en cada fase del diseño y de reglas para pasar de una fase a la siguiente.

El diseño de bases de datos, guiado por los datos, y el del software, guiado por los procesos, deben hacerse de forma coordinada para que el modelo de procesos y el esquema conceptual sean consistentes entre sí y completos (todo dato requerido por los procesos debe estar en el esquema y toda operación requerida por la base de datos debe reflejarse en el modelo de procesos).

Capítulo 2

Modelo Entidad-Relación

Es una familia de modelos propuesta por Peter P. Chen en 1976 con extensiones y aportaciones de muchos otros autores, muy extendida y soportada por herramientas **CASE** (*Computer-Aided Software Engineering*) de bases de datos.

2.1. Entidades

Una **entidad** (*entity*) es aquello del mundo real con existencia propia distinguible del resto, sea esta física o real, o abstracta o conceptual. Un **atributo** (*attribute*) es una propiedad de una entidad, de forma que una entidad particular se describe por los valores de sus atributos.

Un **tipo de entidad** (*entity set*) es un conjunto de entidades con los mismos atributos, y describe el **esquema** o **intensión** para un conjunto de entidades, las **instancias** del tipo, con la misma estructura. Las instancias de un tipo de entidad se agrupan en un **conjunto de entidades** o **extensión**. Representamos un tipo de entidad con un rectángulo que contiene el nombre del tipo.

2.2. Atributos

Un atributo se representa con una elipse que contiene su nombre, y se une con una línea al tipo de entidad que lo tiene. Su **dominio** (*value set*) es el conjunto de valores que puede adoptar.

Un atributo es **simple** si es indivisible o **compuesto** si puede dividirse en otros con significado propio, lo que representamos uniendo los subatributos con el atributo compuesto en vez de con el tipo de entidad. Es **almacenado** si su valor procede del mundo real o **derivado** si se calcula a partir de otra información ya existente, siendo por tanto redundante, y en tal caso la elipse se dibuja punteada.

Es **monovalorado** o **monovaluado** si solo admite un valor para cada instancia en cada momento, o **multivalorado** o **multivaluado** si puede tener más de un valor a la vez para la misma entidad, en cuyo caso la elipse es doble.

Un atributo es **opcional** si su valor puede ser **nulo** (*null value*), que indica que el valor existe pero no se conoce, no se sabe si existe o la entidad no tiene un valor aplicable para este.

La **cardinalidad** de un atributo es un par (m, n) donde $0 \leq m \leq n \leq +\infty$, $m < +\infty$, que indica que una instancia del padre puede tener un mínimo de m y un máximo de n (o no tener máximo si $n = +\infty$) para el atributo. Se indica añadiendo el par a la línea que conecta al atributo con el padre, cambiando $+\infty$ por una letra, siendo los valores por defecto $(1, 1)$ para atributos monovalorados o $(1, +\infty)$ para multivalorados.

Un atributo **clave** o **identificador** es uno con valor distinto para cada entidad, y se indica subrayando el nombre. Una **clave compuesta** es un conjunto mínimo de atributos que, juntos, actúan como una clave, y se indica añadiendo un punto en cada línea que une un elemento del conjunto con su padre, uniendo estos puntos y haciendo esta unión el padre de un nuevo atributo con nombre subrayado.

Si hay varias claves, las llamamos **identificadores candidatos** (ICs) y, de entre ellas, elegimos uno como **identificador principal** (IP), que indicamos añadiendo «IP» a la línea que lo une con el padre, y llamando al resto **identificadores alternativos**.

2.3. Relaciones

Un **tipo de relación** es una estructura genérica o abstracción de un conjunto de **relaciones** o **interrelaciones** (*relationship*), asociación entre varias entidades representadas por el tipo de relación y una n-upla de entidades con $n \geq 2$, donde las tuplas del mismo tipo tienen igual tamaño y cada posición de estas tiene valores de un mismo tipo de entidad y cumple un mismo **rol** o **papel** en la relación.

Un tipo de relación se representa por un rombo que contiene el nombre del tipo con una línea de unión al tipo de entidad por cada elemento de las relaciones que contiene. El rol se puede indicar en esta línea mediante su nombre o se puede inferir del orden de lectura de arriba a abajo y de izquierda a derecha según dónde se hayan conectado las líneas.

El **grado** de un tipo de relación es el número de tipos de entidad que participan. Una relación es **binaria** si su tipo de relación es de grado 2, **ternaria** si es de grado 3, **cuaternaria** si es de grado 4 y **reflexiva** o **recursiva** si es de grado 1.

La **cardinalidad** de una posición en un tipo de relación es el número de relaciones que puede haber del tipo en que dicha posición tiene un mismo elemento. Se representa igual que la cardinalidad de atributos, en la línea que une el tipo de relación con el tipo de entidad, y es obligatorio indicarla.

Una relación puede tener atributos pero no claves, y se representan poniendo el tipo de relación como «padre» del atributo.

Un tipo de entidad es **fuerte** si tiene un atributo identificador o **débil** si no lo tiene, en cuyo caso debe estar en una o varias relaciones de cardinalidad $(1, 1)$ con otros tipos de entidad y, si es necesario, incluir una **clave parcial** o **discriminante**, de forma que, dadas dos entidades distintas del tipo, o estas difieren en la entidad fuerte con la que se relacionan para alguna relación, o difieren en la clave parcial.

Las claves parciales pueden ser compuestas, puede haber varias y se representan como los identificadores pero haciendo el subrayado punteado. Para indicar que un tipo de entidad es débil, se rodea de un doble rectángulo y se hacen dobles tanto las líneas que las unen con tipos de relación con entidades que participan de la identificación como los rombos que representan estos tipos de relación.

A veces una herramienta de diseño solo permite relaciones binarias, en cuyo caso podemos representar un tipo de relación entre más entidades como un tipo de entidad débil con una

relación binaria identificadora por cada posición de las relaciones.

2.4. Tipos de relación excluyentes

Dos o más tipos de relación son **exclusivos** respecto a una posición de cada uno con un mismo tipo de entidad si cada entidad de dicho tipo solo puede participar en uno de los tipos de relación en dicha posición. Esto se representa con una curva que corta con todas las uniones entre el tipo de entidad y los tipos de relación para dichas posiciones.

Un tipo de entidad puede ser **subtipo** de otro, creando una **jerarquía** entre tipos de entidad que se puede formar por **especialización**, partiendo del tipo general y creando a los específicos, o por **generalización**, partiendo de los específicos y creando el general. El tipo de entidad general es el **supertipo** de los específicos.

En general los subtipos están definidos según una característica distintiva o **discriminante** de las entidades del supertipo. Este se representa mediante un círculo sin nombre, que puede tener atributos y que se une con una línea al supertipo y con una línea a cada subtipo cortada por una línea curvada con los extremos hacia el discriminante. Un subtipo puede tener atributos propios específicos y participar en relaciones, y **hereda** todos los atributos del supertipo y relaciones en las que participa.

Para un cierto discriminante, los subtipos son **solapados** (*overlapped*) si una instancia del supertipo puede serlo de más de un subtipo, opción por defecto que puede representarse si se quiere con una «o» dentro del círculo, o **disjuntos** si una instancia del supertipo puede serlo de un solo subtipo como máximo, lo que se representa con una «d» dentro del círculo.

La especialización puede ser **parcial** si puede haber instancias que no pertenecen a ninguno de los subtipos, la opción por defecto, o **total** o **completa** si toda instancia del supertipo debe serlo también de uno de los subtipos, lo que se representa haciendo doble la línea que une el supertipo con el discriminante.

Capítulo 3

Diseño conceptual

Un **esquema conceptual** es una descripción del contenido de una base de datos que persigue entender su estructura, semántica, relaciones y restricciones, independientemente de aspectos de implementación, usando un modelo de datos de alto nivel, más expresivo y general que uno de representación, para la comunicación entre usuarios, diseñadores y analistas. Consta de un **diagrama entidad-relación** y un **diccionario de datos**.

El diseño conceptual es el proceso de construir un esquema conceptual a partir de un análisis meticuloso del **catálogo de requisitos de datos** mediante refinamiento y estructuración progresivos. El diseño puede ser:

- **Centralizado**: El equipo de diseño de la base de datos reúne los requisitos de cada grupo de usuarios, aplicación o subsistema, decide cómo combinarlos, diseña el esquema conceptual y especifica los esquemas externos.
- **Integración de vistas**: Cada grupo de usuarios diseña un esquema conceptual, y el equipo de diseño de la base de datos integra estas vistas en un esquema global.

Es recomendable seguir una metodología estructurada en que se trabaje interactivamente con los usuarios tanto como sea posible, siguiendo un enfoque guiado por los datos que combine técnicas para conceptualizar, normalizar y validar los datos e incorpore consideraciones estructurales y de integridad; usando diagramas para representar los modelos y un lenguaje de diseño de bases de datos para representar semántica adicional que no pueda expresarse en los diagramas; construyendo un diccionario de datos para complementar los diagramas, y repitiendo pasos de la metodología siempre que sea necesario.

Pasos en el diseño:

1. Identificar tipos de entidad. Criterios:

   a) Lingüístico: Suelen ser sujetos o complementos directos. Los nombres propios suelen ser instancias.

   b) De categorización de objetos: Son conceptos con más propiedades que su nombre y que describen un tipo de objetos con existencia autónoma.

   Se asigna un nombre significativo a cada tipo de entidad y se registra en el diccionario de datos junto a su descripción y sinónimos si hay.

2. Identificar tipos de relación. Criterios:

   a) Lingüístico: Suelen ser verbos, locuciones verbales, preposiciones o locuciones preposicionales entre nombres de entidad (instancias o tipos). Del uso de tipos de entidad en singular o plural se obtiene la cardinalidad.

   b) De categorización: Proporcionan un vínculo entre entidades que hace posible la selección de una entidad a través de una referencia a una propiedad de otra.

   Solo se incluyen las relaciones de interés para el usuario. La mayoría relacionan dos entidades, aunque pueden relacionar más. Las relaciones se distinguen por la tupla de entidades relacionadas, por lo que si hace falta un atributo para distinguirlas, este se ha de convertir en entidad.

   Se asigna un nombre significativo a cada tipo de relación y se registra en el diccionario de datos junto a su descripción, sinónimos si los hay y, para cada entidad relacionada, el tipo de entidad, la cardinalidad y, si es necesario, el rol.

3. Identificar y asignar atributos a los tipos de entidad y relación. Criterios:

   a) Lingüístico: Como para los tipos de relación, pero entre un nombre de entidad y una propiedad.

   b) Categorización: Son conceptos simples con un valor y sin propiedades asociadas.

   Se usa una entidad si el concepto tiene asociados otros atributos de interés o está relacionado con más entidades, o atributos si solo tiene su valor y no participa claramente en vínculos.

   Un atributo es compuesto si resulta natural dar nombre a un grupo de propiedades simples y se referencia tanto el conjunto como unidad como sus partes por separado, es simple si solo se referencia como unidad, y es un conjunto de atributos simples si las propiedades son independientes y solo se referencian individualmente. Es opcional si puede no estar y derivado si puede obtenerse a partir de otros.

   Dentro de cada tipo de entidad o relación, se asigna un nombre significativo a cada atributo y se registra en el diccionario de datos, clasificado por tipo al que pertenece, junto con su descripción, sinónimos si los hay, tipo de datos, subatributos si el atributo es compuesto, fórmula de cálculo si es derivado, y otras propiedades como si es opcional o no.

4. Determinar atributos clave. Establecer los atributos identificadores de cada tipo de entidad y, si un tipo de entidad tiene varios, elegir un identificador principal o **clave primaria** buscando que tenga menos atributos, valores de menor longitud o sea más usado por los usuarios. El resto quedan como alternativas.

   Si una entidad no puede identificarse por sí misma, podría ser débil. El verbo tener puede relacionar un tipo de entidad fuerte con uno débil que depende de él o un tipo de entidad con sus atributos.

5. Considerar restricciones de integridad. Documentar las restricciones necesarias para que los datos no queden incompletos, imprecisos o incoherentes, redactando **reglas de integridad** en lenguaje natural que no pueden expresarse en el diagrama entidad-relación.

6. Comprobar redundancia. El esquema conceptual debe ser mínimo. Una relación es **redundante** si se puede obtener la misma información mediante otras relaciones. No debe haber elementos sinónimos, relaciones redundantes ni entidades «colgantes». Se debe considerar si hay que modelar los estados anteriores de cierta información o solo el actual.

7. Validar el esquema conceptual contra las transacciones de usuario. Un **catálogo de requisitos de transacciones** contiene los tipos de transacciones de entrada, actualización, eliminación y consulta de datos que requiere el usuario. Para asegurarse de que todas están soportadas, se dibuja sobre el diagrama el camino tomado por cada transacción, lo que permite visualizar qué áreas del modelo son cruciales; cuáles no son necesarias y se pueden eliminar, y cuáles se han omitido o definido mal. Como esto complica los diagramas, se suelen usar varias copias. Finalmente se documenta el camino tomado por cada transacción del catálogo.

8. Revisar el esquema conceptual con el usuario. Ver que este considera el modelo como una verdadera representación de los requisitos. Si hay anomalías, se hacen cambios y se repite el proceso, hasta que el usuario autorice el esquema.

Capítulo 4

Modelo relacional

Introducido por Ted Codd en 1970 desde IBM Research, el **modelo relacional** es un modelo de representación de datos lógico basado en registros y el más usado en procesamiento de datos convencional.

Un **dominio** es el conjunto de valores atómicos que puede tomar un atributo. Si el SGBD soporta dominios, puede detectar errores como el de intentar comparar atributos de distintos dominios.

Una **relación** $S : \{D_i\}_{i \in I}$ está compuesta por un **esquema** o **cabecera** $\{D_i\}_{i \in I}$, donde I es un conjunto finito de atributos y D_i es el dominio del atributo i, y un **estado**, **cuerpo** o **instancia** $S \subseteq \prod_{i \in I} D_i$ finito, cuyos elementos son **tuplas**. El estado de una relación varía según el tiempo de forma discreta (los tiempos en los que cambia son puntos aislados), mientras que el esquema puede variar en el tiempo de forma discreta pero no suele hacerlo. Llamamos **cardinalidad** de una relación al número de tuplas en su estado y **grado** al número de atributos.

Podemos representar un **esquema** de relación como *RELACIÓN*`(`*atributo*`, ...)` o mediante una tabla de la forma

RELACIÓN	*atributo*	...

.

Representamos el esquema mediante una tabla con una columna por atributo y una fila por tupla. Esta representación es muy sencilla y fácil de utilizar y entender, pero da a entender falsamente un orden entre las filas y columnas de la tabla.

Una **base de datos relacional** es un par que varía con el tiempo de forma discreta formado por una familia finita de relaciones y una serie de **reglas** o **restricciones de integridad**, proposiciones lógicas que debe cumplir la base de datos para que los datos tengan sentido. El esquema de la base de datos está formado por la familia de los esquemas de las relaciones y las reglas de identidad, y el estado es la familia de los estados de las relaciones.

Principio de información: Todo dato en una base de datos relacional se expresa únicamente como valor explícito en una posición de columna dentro de una fila en una tabla.

Existe un valor **nulo** (*null*), que puede estar o no en el dominio de un atributo, que representa información perdida, ausencia de información o valor no aplicable a cierto atributo. Este no se trata como valor, sino como indicación de que el valor real del atributo es desconocido, por lo que dos valores nulos no se consideran iguales.

Dada una relación R con conjunto de atributos I, una **clave** de R es un conjunto de atributos $J \subseteq I$ que cumple la **restricción de unicidad**, consistente en que dos tuplas distintas de R

deben diferir en alguno de los atributos de J, y la **restricción de irreductibilidad**, que dice que no existe un $K \subsetneq J$ tal que K cumpla la restricción de unicidad. Una clave J es **simple** si $|J| = 1$ y **compuesta** si $|J| > 1$. A cada clave le corresponde una regla de integridad.

R debe tener al menos una clave, pues al menos I cumple la restricción de unicidad y es finito. Llamamos **claves candidatas** a las claves de R, **clave primaria** (*primary key*, **PK**) a la clave candidata que elegimos para identificar a las tuplas de R y **claves alternativas** (*alternative keys*, **AK**) al resto de claves candidatas. **Restricción de integridad de entidad:** ningún atributo de una clave primaria puede valer nulo.

La interrelación o vínculo de una relación R a una R' se hace incluyendo en R una **clave ajena**, **externa** o **foránea** (*Foreign Key*, **FK**). Así, si $\{D_i\}_{i \in I}$ es el esquema de R, $\{E_j\}_{j \in J}$ es el de R' y $P \subseteq J$ es la clave primaria de R', una clave ajena en R hacia R' es un conjunto $F \subseteq I$ de atributos para los que existe una biyección $\sigma : F \to P$ con $E_{\sigma(f)} = D_f \setminus \{\texttt{NULL}\}$ para $f \in F$ junto con una **restricción de integridad referencial** que afirma que, para toda tupla T en el estado de R, o T_f es nulo para todo $f \in F$, o existe a la vez una tupla U en el estado de R' tal que $\{T_f\}_{f \in F} = \{U_{\sigma(f)}\}_{f \in F}$. El primer caso, si se permite, indica que esa fila no participa en el vínculo entre relaciones.

Una clave ajena es simple o compuesta según lo sea la clave primaria a la que referencia, y puede ser o formar parte de la clave primaria (para representar entidades débiles) o de una alternativa. Si la clave ajena referencia a su misma tabla, hablamos de una **auto-referencia**.

Un **diagrama referencial** indica gráficamente la existencia de claves ajenas mediante la representación por tablas de los esquemas de las relaciones de la base de datos junto con una flecha de cada clave ajena a la clave primaria correspondiente.

Un **camino referencial** es un recorrido del diagrama referencial para obtener información relacionada, y si este empieza y acaba en la misma relación hablamos de un **ciclo referencial**.

El SGBD se encarga de mantener la integridad referencial rechazando operaciones que rompan la integridad o ejecutando acciones adicionales que la restauren. Las operaciones de crear una tupla con un valor de clave ajena que no se corresponde con ninguno de la clave primaria correspondiente, o modificar una tupla existente para que ocurra esto, siempre se rechazan. En otros casos el diseñador del esquema especifica la acción a realizar:

- Al borrar una tupla referenciada por otra:
  - Cascada (`CASCADE`): borrar también la tupla que hacía referencia, lo que puede desencadenar más borrados.
  - Establecer la clave ajena a nulo (`SET NULL`), solo si esta lo permite.
  - Rechazar la operación (`NO ACTION`). Si el borrado se hacía como resultado de una cascada, la operación entera se rechaza.
- Al modificar la clave primaria de una tupla referenciada por otra:
  - Cascada: establecer la clave ajena al nuevo valor, lo que puede desencadenar más cambios.
  - Establecer la clave ajena a nulo, solo si esta lo permite.
  - Rechazar la operación. Si el cambio era resultado de una cascada, la operación entera se rechaza.

Las actualizaciones de una base de datos son siempre **atómicas**: o se modifica todo lo necesario o no se hace ningún cambio.

Capítulo 5

Diseño lógico

El **diseño lógico** es la transformación del esquema conceptual en un **esquema lógico**, eliminando redundancias, evitando cargas suplementarias y maximizando la simplicidad para conseguir una estructura lógica adecuada y un equilibrio entre los requisitos de usuario y la eficiencia. Para conseguir la máxima portabilidad, se introduce el SGBD específico de forma tardía, permitiendo implementar el esquema lógico en distintos SGBD y migrar entre SGBDs.

1. Se empieza con un **diseño lógico estándar** (**DLS**), eligiendo el **modelo lógico de datos estándar** (**MLS**), independiente del SGBD, y se describe en este un **esquema lógico estándar** (**ELS**), que en el modelo relacional se puede hacer con pseudolenguaje o SQL.

2. Se elige el SGBD y se hace el **diseño lógico específico** (**DLE**), en que se adapta el ELS a un **esquema lógico específico** (**ELE**), que se describe en el lenguaje del SGBD específico (para bases de datos relacionales, MySQL, PostgreSQL, etc.) y usa su modelo de datos concreto.

El primer paso del diseño lógico relacional es obtener las tablas, que en pseudolenguaje se definen como:
NOMBRE_TABLA (***atributo***, ...)
Admiten NULL: *atributo*, ...
Clave Primaria: *atributo*, ...
Claves Alternativas (UNIQUE): *(lista numerada de atributos o tuplas de atributos)*
Claves Ajenas (FOREIGN KEY):
(lista numerada de «(*atributo*, ...) Referencia_a *NOMBRE_TABLA*(*atributo_clave*, ...)»)
Derivado:
(lista numerada de «*atributo* = *fórmul*a»)
Comprobar:
(lista numerada de restricciones)

Para cada tipo de entidad, se crea una tabla de nombre igual o muy similar, con una columna por cada atributo simple monovaluado, sea del tipo de entidad o, recursivamente, de un atributo compuesto monovaluado de este. Las columnas correspondientes a atributos admiten nulo si y sólo si los atributos correspondientes o los atributos compuestos a los que pertenecen, recursivamente, lo admiten.

Un atributo multivalorado se trata como una entidad débil asociada al tipo de entidad o atributo padre al que pertenece, o si se puede, como una entidad fuerte relacionada con este. El atributo tendría cardinalidad $(1,1)$ en esta relación, y el padre, la cardinalidad del atributo.

La clave primaria corresponde al identificador principal y las alternativas al resto. Si el tipo de entidad es débil, la clave primaria contendrá al identificador principal parcial, pero tanto a esta como a las alternativas se les añadirán claves ajenas al modelar las relaciones.

Para cada relación:

1. Si no hay elementos de cardinalidad $(1,1)$ o $(0,1)$, o si la relación es recursiva con cardinalidad $(0,1)$ a ambos lados, se crea una tabla para la relación con los atributos simples monovaluados de la relación y una clave ajena por cada elemento relacionado.

   La clave primaria es el conjunto de claves ajenas o, si es posible, un subconjunto de este. Si el conjunto de claves ajenas no es clave de la nueva relación, se ha cometido un error en el diseño conceptual que normalmente se puede subsanar añadiendo algún atributo a la clave. Las cardinalidades de elementos distintas a $(0,n)$ se traducen como **restricciones de integridad generales** o **asertos**.

2. Los elementos de tipos distintos con cardinalidad $(1,1)$ se combinan en una tabla cuyas columnas son los atributos simples monovaluados de los tipos de entidad y la relación. La clave primaria se elige de entre las primarias de cada tipo de entidad fuerte.

3. Si tras esto hay un elemento con cardinalidad $(1,1)$, se añaden a este claves ajenas que no admiten nulo al resto de elementos que quedan en la relación, si los hay. En tal caso se llama **entidad hijo** al elemento de cardinalidad $(1,1)$ y **entidades padre** al resto.

   Si la entidad hijo era débil y la relación formaba parte de su identificación, las claves ajenas se añaden a la clave primaria. Si las entidades padre no tienen cardinalidad $(0,n)$, esta se modela con un aserto. Si una entidad padre tiene cardinalidad $(0,1)$, la clave ajena correspondiente en el hijo es clave alternativa.

4. Si no hay elementos con cardinalidad $(1,1)$ pero sí $(0,1)$, se elige uno de cardinalidad $(0,1)$ como entidad hijo y se opera como en el caso anterior, salvo que las claves ajenas y atributos de relación admiten nulo.

5. Las relaciones recursivas con cardinalidad $(0,1)$ a un lado y $(1,1)$ al otro se pueden modelar como entidades padre e hijo o creando una nueva tabla.

Para indicar que cada entidad de un cierto tipo solo puede participar en uno de entre varios roles de tipos de relación, si la entidad se ha modelado como hijo en todas estas relaciones, se añade una **restricción de comprobación** en la tabla correspondiente al tipo de entidad, y de lo contrario se añade un aserto.

Finalmente, se traducen las restricciones del diccionario de datos no representadas en el diagrama conceptual.

Los siguientes pasos del diseño lógico son:

1. Garantizar que las tablas tienen un conjunto mínimo de atributos pero suficiente para soportar los requisitos de datos y con redundancia mínima, usando **técnicas de normalización**. Se usan las **reglas de las formas normales**:

1FN. Todo atributo (relacional) representa un valor atómico, no uno multivalorado o compuesto.

2FN. Cada atributo no clave depende totalmente de una clave, no solo de parte de esta.

3FN. No existen dependencias funcionales transitivas entre los atributos, es decir, ningún atributo depende del valor de otro que no sea clave.

Se recomienda que cada relación sea al menos 3FN.

2. Validar las relaciones contra las transacciones de usuario, intentando «ejecutar manualmente» las transacciones. Si alguna no se puede resolver, se ha cometido algún error en el diseño lógico.

3. Comprobar y documentar las restricciones que evitan que la base de datos quede incompleta, imprecisa o incoherente, evitando la pérdida de semántica en el proceso de traducción. Comprobar:

   a) Que las columnas traducidas de atributos admiten nulo si y sólo si el atributo original lo admite.

   b) Que los atributos solo pueden tener valores legales, especialmente cuando el dominio no se corresponde exactamente con un tipo de datos de la base de datos.

   c) Que ninguna clave primaria puede contener nulo.

   d) Que las claves ajenas tienen tantas columnas como la clave primaria a la que referencian, que son de tipo correcto, que está bien indicado si admiten nulo o no y que las acciones de mantenimiento de la integridad referencial son las adecuadas según su semántica.

   e) Que las reglas de integridad generales están representadas como comprobaciones o asertos.

4. Revisar con los usuarios el esquema lógico y el diccionario de datos, que deben estar completos y documentados.

5. Considerar el crecimiento futuro. Es importante que el esquema lógico pueda evolucionar con efecto mínimo para los usuarios, aunque esto puede ser muy difícil de conseguir si no se sabe qué cambios se desea realizar en un futuro y solo considerarlo puede resultar muy caro en tiempo y dinero. En general, los cambios considerados consisten en ampliar cardinalidades.

Finalmente, para el diseño lógico específico, se estudia la correspondencia entre los conceptos del diseño lógico estándar y los del del SGBD. Si este soporta el modelo estándar sin restricciones (**soporte total**), la transformación es casi directa, mientras que si no soporta algunos conceptos o lo hace pero con limitaciones (**soporte parcial**), se deben usar conceptos alternativos o programación complementaria.

Capítulo 6

SQL

SQL (*Structured Query Language*, lenguaje estructurado de consulta) es un el primer lenguaje de bases de datos relacionales de alto nivel. Fue diseñado e implementado en los años 70 en el *IBM Research Laboratory* en San José, California, para el SGBD relacional System R.

Es un estándar **ANSI** (*American National Standards Institute*) e **ISO** (*International Standardization Organization*). La primera versión del estándar fue SQL1 o ANSI 1986, del que hubo una revisión en 1989 llamada SQL-89. Después se creó SQL2 o SQL-92. SQL:1999 incluye extensiones de orientación a objetos, disparadores (también llamados *triggers* o reglas activas), etc., y SQL:2003 incluye conceptos como XML.

Nos centramos en SQL-92. Hay 3 niveles de compatibilidad con el estándar: *Entry SQL*, *Intermediate SQL* y *Full SQL*, y los proveedores suelen incluir características no estandarizadas. El código SQL se puede incorporar dentro de código en un lenguaje de programación de propósito general.

Los comandos se organizan en:

- **Lenguaje de definición de datos** (**LDD**): Para crear, modificar o eliminar estructuras de datos como tablas o vistas.
- **Lenguaje de manipulación de datos** (**LMD**): Para introducir, actualizar, eliminar y consultar datos.

En SQL, las tablas no contienen un conjunto de filas sino un **multiconjunto** o **saco** (*bag*), en el que puede haber varias filas idénticas. Las columnas están ordenadas por orden de creación, y una clave ajena puede referenciar a una alternativa.

Se describe un subconjunto de la gramática de SQL, en ABNF y omitiendo los espacios en blanco por simplicidad[1].

```
sql-statement = (ddl-stmt / dml-stmt / tcl-stmt) ";"
id = ALPHA*(ALPHA|DIGIT|"_")
sid = [id "."] id
uint = 1*DIGIT
expr = number-expr / string-expr
```

[1]En concreto, si `W = (WSP / CR / LF)` y `ALNUM = (ALPHA / DIGIT / "_")`, entonces hay `*W` entre dos términos que, en el código, se separaran por espacio, o `1*W` si, además, alguno de los términos está en `1*ALNUM`.

```
number-expr = term *(("+" / "-") term)
term = primary-expr *(("*" / "/") primary-expr)
string-expr = primary-expr 1*("||" primary-expr)
primary-expr = literal / [sid "."] id / set-function / "(" query
    ")" / "(" expr ")"
literal = number-literal / string-literal
number-literal = ["+"/"-"]uint["."uint]
string-literal = "'"*(nonquote-character/"''")"'"
```

`||` es la concatenación de cadenas, y la secuencia `''` en una cadena indica una `'`.

6.1. Transacciones

```
tcl-stmt = ("COMMIT" / "ROLLBACK") ["WORK"]
```

Una **transacción** es la unidad lógica de procesamiento, una secuencia de operaciones realizadas sobre una base de datos de forma que, si una falla, la base de datos vuelve a su estado anterior al comienzo de la transacción. En SQL, toda operación se hace dentro de una transacción. `COMMIT` confirma los cambios de la transacción actual, pasando a una nueva transacción, y `ROLLBACK` deshace todos los cambios de la transacción actual.

6.2. Lenguaje de definición de datos

```
ddl-stmt = schema-def / drop-schema-stmt / table-def /
    alter-table-stmt / drop-table-stmt / view-def /
    drop-view-stmt / assertion-def / drop-assertion-stmt /
    set-constraints-stmt
```

6.2.1. Esquemas

Un **esquema** agrupa tablas, vistas, dominios, permisos o privilegios, asertos, etc., como los del mismo usuario o aplicación.

```
schema-def = "CREATE" "SCHEMA" id "AUTHORIZATION" id
```

El segundo `id` indica el usuario o cuenta propietario del esquema. En `sid`, se puede indicar un esquema **explícito** indicando *esquema*`.`*elemento* o un esquema **implícito** como *elemento*, donde se supone el esquema activo en la cuenta del usuario actual.

```
drop-schema-stmt = "DROP" "SCHEMA" id [drop-behavior]
drop-behavior = "CASCADE" | "RESTRICT"
```

Elimina un esquema. `RESTRICT`, el `drop-behavior` por defecto, destruye el esquema solo si no contiene ningún elemento, y `CASCADE` destruye también todos los elementos del esquema.

Desde SQL-92, los esquemas se organizan en **catálogos**, conjuntos de todos los esquemas de base de datos en el mismo entorno, con un esquema especial, `INFORMATION_SCHEMA`, que almacena la definición de todos los elementos de todos los esquemas del catálogo, de modo que

se pueden compartir elementos entre esquemas del mismo catálogo pero solo pueden definirse restricciones de integridad referencial entre tablas de un mismo catálogo.

6.2.2. Crear una tabla

```
table-def = "CREATE" "TABLE" sid "(" table-elements ")"
table-elements = table-element *("," table-element)
table-element = column-def / table-constraint
column-def = id type [default-clause] *column-constraint
```

Un `create-table-stmt` crea una tabla con una serie de columnas, cada una con el tipo indicado, que quedan en orden de aparición. Las filas no están ordenadas salvo que la tabla sea `HEAP`, en cuyo caso quedan en orden de inserción.

```
type = number-type / string-type / time-type
number-type = "INTEGER" / "INT" / "SMALLINT" / "REAL" / "DOUBLE"
    "PRECISION" / "FLOAT" ["(" uint ")"] / ("NUMERIC" /
    "DECIMAL" / "DEC") ["(" uint ["," uint] ")"]
string-type = ("CHAR" / "BIT") ["(" uint ")"] / ("VARCHAR" /
    "BIT VARYING") "(" uint ")" / "NCHAR" ["(" uint ")"]
time-type = "DATE" / ("TIME" / "TIMESTAMP") ["WITH" "TIME"
    "ZONE"] / "INTERVAL" ("YEAR" ["(" uint ")"] "TO" "MONTH" /
    "DAY" ["(" uint ")"] "TO" "SECOND" ["(" uint ")"])
```

`INTEGER` o `INT` y `SMALL INT` son tipos enteros. `REAL` indica punto flotante de precisión simple, `DOUBLE PRECISION` de doble precisión, y `FLOAT(`*p*`)` de precisión *p*. `NUMERIC`, `DECIMAL` o `DEC` recibe como parámetros opcionales el número de dígitos a almacenar y, opcionalmente, un factor de escala. `CHAR` y `BIT` indican una cadena de caracteres o bits de una cierta longitud fija, por defecto 1. `VARCHAR` y `BIT VARYING` son similares, pero pueden contener elementos de menor longitud. `DATE` indica una fecha en formato `YYYY-MM-DD`; `TIME` una hora en formato `hh:mm:ss`; `TIMESTAMP` una marca de tiempo con fecha, hora, fracción de segundo y, si se incluye `WITH TIME ZONE`, una zona horaria.

`INTERVAL` indica un periodo de tiempo. Los intervalos `YEAR TO MONTH` contienen los parámetros año y mes, con el tamaño opcionalmente indicado para el año, y los `DAY TO SECOND` contienen día, hora, minuto y segundo, con el tamaño opcionalmente indicado para el día y la fracción de segundo.

```
column-constraint = (["NOT"] "NULL" / "PRIMARY" "KEY" / "CHECK"
    "(" bool-expr ")" / fk-clause) [constraint-attribute]
default-clause = "DEFAULT" (literal / "NULL")
```

En las restricciones de columna, `NULL` indica que la columna puede contener `NULL`, y `NOT NULL` que no. Por defecto se asume `NULL` salvo para las componentes de una clave primaria, para las que se asume `NOT NULL`. El resto de restricciones equivalen a las restricciones de tabla correspondientes, sin nombre y con lista de columnas solo con la columna actual.

Cuando se crea una fila en la tabla sin indicar el valor de una columna, se usa el valor por defecto, indicado en la `default-clause`, o si esta no aparece, `NULL`, dando error si la columna era `NOT NULL`.

```
table-constraint = ["CONSTRAINT" sid] (("PRIMARY" "KEY" /
    "UNIQUE") column-list / "CHECK" "(" bool-expr ")" /
    "FOREIGN" "KEY" column-list fk-clause) [constraint-attribute]
column-list = "(" id *("," id) ")"
```

Con `CONSTRAINT`, se puede dar un nombre a una restricción, que se guarda en el espacio de nombres global del esquema junto con los nombres de tablas y vistas para poder manejar la restricción en el futuro.

Una **restricción de clave candidata** `PRIMARY KEY` o `UNIQUE` indica que las columnas en la lista forman una clave primaria o alternativa, respectivamente. Una **de comprobación** `CHECK` indica que, para cada fila de la tabla, la expresión, en la que cada identificador de columna se refiere al correspondiente valor en la fila, evalúa a verdadero.

```
fk-clause = "REFERECES" sid column-list *("ON" ("DELETE" /
    "UPDATE") ("SET" "NULL" / "SET" "DEFAULT" / "CASCADE" / "NO"
    "ACTION"))
```

Una **restricción de clave ajena** o **externa** `FOREIGN KEY` indica que las columnas en la primera lista forman una clave ajena hacia la tabla indicada justo tras `REFERENCES`, concretamente a las columnas de la tabla en la segunda lista de columnas, de igual tamaño y en el mismo orden, que deben formar una clave primaria o alternativa.

Las cláusulas `ON DELETE` y `ON UPDATE` indican qué hacer si se borra la fila a la que hace referencia la clave ajena o si se modifica el valor de la clave primaria o alternativa. `SET NULL` establece el valor de la clave ajena a `NULL`, `SET DEFAULT` lo establece a su valor por defecto, `CASCADE` hace el mismo cambio en la fila con la clave ajena (borrándola o cambiando el valor de la clave ajena), y `NO ACTION` produce un fallo.

6.2.3. Modificar una tabla

```
alter-table-stmt = "ALTER" "TABLE" sid [alter-column /
    alter-constraint]
alter-column = "ADD" column-def / "ALTER" id ("SET"
    default-clause / "DROP" "DEFAULT") / "DROP" "COLUMN" id
    drop-behavior
alter-constraint = "ADD" table-constraint / "DROP" "CONSTRAINT"
    sid drop-behavior
```

Permite añadir columnas, modificar su valor por defecto o eliminarlas, y añadir o eliminar restricciones. Una columna no se puede eliminar si una restricción hace referencia a ella, pero si se indica `CASCADE`, se eliminan también estas restricciones.

6.2.4. Eliminar una tabla

```
drop-table-stmt = "DROP" "TABLE" sid [drop-behavior]
```

Elimina las filas y la definición. Sin `CASCADE`, solo se elimina la tabla si ninguna restricción de integridad externa le hace referencia.

6.2.5. Asertos

La información almacenada debe cumplir en todo momento las reglas existentes. Una regla de integridad se compone de:

1. Un nombre, que aparece en los mensajes de error por intentar incumplir la regla.
2. Una expresión booleana que indica cuándo se satisface la regla.
3. Una respuesta a un intento de incumplimiento. Por defecto es fallar con un mensaje de error, pero podría ser un procedimiento de complejidad arbitraria.

Las reglas de identidad son parte de los metadatos, y el **subsistema de integridad** controla que estas no se incumplan. Una base de datos está en un **estado de integridad** o es **correcta** si no incumple ninguna regla de identidad. Hay 3 tipos de reglas:

1. **De dominio**: Define el dominio de una columna.
2. **De tabla**: Incluida en la definición de una tabla.
3. **Generales** (**asertos**): Al mismo nivel que una tabla o vista, no incluidas en la definición de ninguna tabla.

Ya hemos visto las de dominio y las de tabla.

```
assertion-def = "CREATE" "ASSERTION" sid "CHECK" "(" bool-expr
    ")" [constraint-attribute]
drop-assertion-stmt = "DROP" "ASSERTION" sid
```

En la condición de un aserto, se suele usar que $\forall x, P(x) \equiv \nexists x : \neg P(x)$. No se puede crear una regla de identidad si el estado actual de una base de datos no la cumple.

La mayoría de SGBDs no soportan asertos, y en su lugar se usan formas no declarativas de expresar estas restricciones como procedimientos o funciones almacenados y *triggers*.

6.2.6. Vistas

```
view-def = "CREATE" "VIEW" sid [column-list] "AS" query ["WITH"
    "CHECK" "OPTION"]
```

Una **vista** es una presentación de datos de una o más **tablas base** (otras tablas o vistas) como si fuera una tabla. No contiene información, sino que refleja el estado de las tablas base. Las vistas permiten almacenar consultas complejas, simplificar las sentencias, presentar los datos con otra perspectiva distinta a la de las tablas base, proporcionar seguridad (restringiendo el acceso a las tablas base forzando a acceder a ellas a través de vistas con información limitada), ocultar la complejidad y aislar a las aplicaciones de los cambios en la definición de las tablas base.

La `column-list` permite dar nuevos nombres a las columnas de la vista, siendo esto obligatorio cuando alguna columna de la consulta no se expresa como nombre de columna de una tabla base. Si una vista se define con `SELECT *` y se añaden columnas a las tablas base, estas no se añaden a la vista.

Una vista puede aparecer en una consulta (`dql-stmt`) como una tabla. Es **actualizable** si la `query` es un `select-stmt` sin agrupación que referencia una sola tabla base en la `from-clause` y contiene una clave candidata de dicha tabla en las `derived-column`. Solo se puede usar un `dml-stmt` distinto de `dql-stmt` en una vista si esta es actualizable, en cuyo caso la sentencia se traduce a una equivalente sobre la tabla base.

Al insertar o actualizar filas de la tabla base desde una vista, puede que estas no aparezcan en la vista por no cumplirse la **condición de definición**, la condición de la `query` que indica si una fila de la tabla base aparece o no en la vista. `WITH CHECK OPTION` hace que un `insert-stmt` o `update-stmt` sobre la vista falle si esto ocurre.

```
drop-view-stmt = "DROP" "VIEW" sid [drop-behavior]
```

Elimina una vista. Sin `CASCADE`, solo la elimina si ninguna regla de integridad le hace referencia.

6.2.7. Modos de comprobación de restricciones

```
constraint-attribute = ["INITIALLY" ("DEFERRED" / "IMMEDIATE")]
   [["NOT"] "DEFERRABLE"]
```

Una restricción puede estar en modo `IMMEDIATE`, comprobándose inmediatamente después de cada sentencia, o `DEFERRED` (**diferida**), comprobándose al final de la transacción. `INITIALLY` indica el modo de la restricción al comienzo de cada transacción, que por defecto es `IMMEDIATE`, y `DEFERRABLE` indica si la restricción es **diferible**, esto es, si puede pasar a modo `DEFERRED`. Si el modo inicial es `IMMEDIATE`, por defecto la restricción es `NOT DEFERRABLE`, y si es `DEFERRED`, por defecto es `DEFERRABLE` y no se puede indicar `NOT DEFERRABLE`.

```
set-constraints-stmt = "SET" "CONSTRAINTS" (sid *("," sid) /
   "ALL") ("DEFERRED" / "IMMEDIATE")
```

Cambia el modo de una o varias restricciones, siendo `ALL` todas las restricciones diferibles. Pasar una restricción de `DEFERRED` a `IMMEDIATE` falla si la restricción no se cumple. Al comienzo de un `COMMIT` se ejecuta implícitamente `SET CONSTRAINTS ALL IMMEDIATE`.

6.3. Lenguaje de manipulación de datos

```
dml-stmt = insert-stmt / delete-stmt / update-stmt / dql-stmt
```

6.3.1. Inserción de filas

```
insert-stmt = "INSERT" "INTO" sid [column-list] query
```

Inserta las filas resultado de la `query` en la tabla `sid`, equiparando cada columna del resultado, en orden, con la correspondiente de la `column-list` o, de omitirse, de la lista ordenada de columnas de la tabla. Normalmente la `query` es de la forma `VALUES (`*`expr`*`, ...)` para insertar una fila. Cada columna no indicada toma el valor en la `default-clause` de su definición o, en su defecto, `NULL`, fallando si esto no es posible.

6.3.2. Eliminación

```
delete-stmt = "DELETE" "FROM" sid ["WHERE" bool-expr]
```

Borra las filas de la tabla que satisfacen la condición, o todas si no se especifica condición.

6.3.3. Actualización

```
update-stmt = "UPDATE" sid "SET" set-clause *("," set-clause)
    ["WHERE" bool-expr]
set-clause = id "=" (expr / "NULL" / "DEFAULT")
```

En cada fila de la tabla que cumple la condición, o en su defecto, en todas, cada columna `id` se establece, bien al valor de la expresión en la `set-clause`, en la que se puede referenciar a las columnas de la fila, bien a `NULL`, bien al valor por defecto.

6.4. Consulta

El lenguaje de consulta forma parte del lenguaje de manipulación de datos.

```
dql-stmt = query [order-by-clause]
query = query-term *(("UNION" ["ALL"] / "EXCEPT") query-term)
query-term = primary-query *("INTERSECT" primary-query)
primary-query = select-stmt / table-value / "(" query ")"
table-value = "VALUES" row-value *("," row-value)
row-value = expr-or-null / "(" expr-or-null *("," expr-or-null)
    ")" / "(" query ")"
expr-or-null = expr / "NULL"
```

Una consulta construye y devuelve una tabla. `UNION`, `EXCEPT` e `INTERSECT` tratan las tablas devueltas por los operandos como conjuntos de filas y devuelven, respectivamente, su unión, diferencia e intersección. `UNION ALL` hace la unión de multiconjuntos, que incluye repeticiones.

6.4.1. Cláusulas `SELECT` y `FROM`

```
select-stmt = "SELECT" ["DISTINCT"] ("*" / derived-column ("," 
    derived-column)) from-clause [where-clause] [group-by-clause
    [having-clause]]
derived-column = expr [["AS"] id]
from-clause = table-ref *("," table-ref)
table-ref = sid [id] / "(" query ")" id / joined-table
```

`SELECT ... FROM` *`tabla`* devuelve una tabla con una columna para cada *`derived-column`* con nombre *`id`* o, en su defecto, *`expr`*, y que para cada fila de la *`tabla`* contiene una fila con el resultado de evaluar cada expresión *`derived-column`* con el nombre de cada *`columna`* y *`tabla.columna`* establecido a su valor en la fila. `*` equivale a listar todas las columnas de la *`tabla`*, en orden. `SELECT DISTINCT` elimina los duplicados.

Si se indican varias *table-ref*, se usa su producto cartesiano, que contiene, para cada fila de la primera tabla y cada fila de la segunda (si hay más de dos tablas, se usa la asociatividad), una fila que contiene los valores de las columnas de la primera y los de la segunda.

Si hay columnas de igual nombre en dos de las tablas indicadas, se deben referenciar como *tabla.columna*. También se puede indicar un alias para la tabla con *tabla alias*, y entonces no se establece *tabla.columna* sino *alias.columna*, permitiendo usar el producto cartesiano de una tabla consigo misma indicando la misma tabla con distintos alias.

```
joined-table = qualified-join / "(" joined-table ")"
qualified-join = table-ref [join-type] "JOIN" table-ref ["ON"
    bool-expr]
join-type = "INNER" / ("LEFT" / "RIGHT" / "FULL") ["OUTER"] /
    "NATURAL"
```

Una **reunión** o *join* es el uso de varias tablas en `FROM` para obtener el producto cartesiano, y se puede hacer separando las tablas por comas o con `joined-table`.

El `join-type` por defecto es `INNER JOIN`, que toma las filas del producto cartesiano que satisfacen `bool-expr`, la **condición de reunión**, o todas si esta se omite. `OUTER JOIN` es similar, pero cuando para una fila de la tabla a la izquierda (`LEFT`), a la derecha (`RIGHT`) o cualquiera de las dos (`FULL`) no tiene una fila al otro lado con la que la condición de reunión se cumpla, se añade una fila en el resultado que en ese otro lado tiene todos los campos a `NULL`.

`NATURAL JOIN` no incluye cláusula `ON`, asume como condición de reunión que las columnas de igual nombre en ambas tablas coincidan y solo incluye una de las dos columnas de igual nombre en el resultado. Tras esto, no se puede calificar una columna de reunión con el nombre de la tabla original.

6.4.2. Cláusula `WHERE`

```
where-clause = "WHERE" bool-expr
```

Indica que, de la tabla indicada en `FROM`, solo aparezcan en el resultado las filas que cumplen la condición.

```
bool-expr = bool-term *("OR" bool-term)
bool-term = bool-factor *("AND" bool-factor)
bool-factor = "NOT" (predicate / "(" bool-expr ")")
predicate = compare-pred / null-pred / like-pred / in-pred /
    quantified-compare-pred / exists-pred
compare-pred = row-value comp-op row-value
comp-op = "=" / "<>" / "<" / "<=" / ">" / ">="
null-pred = row-value "IS" ["NOT"] "NULL"
like-pred = string-expr ["NOT"] "LIKE" string-expr
```

Las operaciones sobre tuplas (`row-value`) consideran una sola expresión como una tupla de un elemento. `NULL` indica desconocimiento de información, por lo que comparar con `NULL` en un `compare-pred` siempre devuelve `FALSE` o `UNKNOWN`, y en su lugar se usa un `null-pred`. Un `like-pred` compara una cadena de caracteres con otra, donde la de la derecha puede tener caracteres reservados o **comodines**, como `%` que representa una cantidad cualquiera de caracteres o `_` que representa un caracter.

```
in-pred = row-value ["NOT"] "IN" "(" (expr *("," expr)) / query
    ")"
quantified-compare-pred = row-value comp-op ["SOME" / "ANY" /
    "ALL"] "(" query ")"
exists-pred = "EXISTS" "(" query ")"
```

Una subconsulta o **consulta anidada** es una consulta en la `where-clause` de otra consulta. Es **correlacionada** si referencia a valores que dependen de la fila de la consulta en la que se encuentra, lo que debe evitarse porque en general supone ejecutar una consulta por cada fila considerada en la consulta externa.

Un `in-pred` comprueba si un valor o tupla está en cierta lista de valores o tuplas, que puede ser el resultado de una consulta anidada.

Un `quantified-compare-pred` devuelve `TRUE` si en el resultado de la subconsulta alguna fila *`r`* cumple *`row-value comp-op r`* (con `SOME` o `ANY`) o todas lo cumplen (con `ALL`), aunque la mayoría de SGBDs permiten usar una lista `expr *("," expr)` en vez de una subconsulta.

Un `exists-pred` comprueba si la subconsulta ha devuelto alguna fila.

6.4.3. Cláusulas `GROUP BY` y `HAVING`

```
group-by-clause = "GROUP" "BY" [sid "."] id *("," [sid "."] id)
```

Agrupa las filas en las que los valores de las columnas indicadas coinciden.

```
set-function = "COUNT" "(" "*" ")" / set-function-type "("
    ["DISTINCT"] expr ")"
set-function-type = "AVG" / "MAX" / "MIN" / "SUM" / "COUNT"
```

`COUNT(*)` devuelve el número de filas del grupo, y `COUNT(`*`expr`*`)` devuelve el número de filas para las que *`expr`* no es `NULL`. `SUM` calcula la suma de los valores de *`expr`* para cada fila para la que *`expr`* no es `NULL`, y `MAX` y `MIN` hacen lo propio para el máximo y el mínimo. Si aparece `DISTINCT`, se consideran las filas tras eliminar duplicados.

Las `set-function` o **funciones de agregados** pueden aparecer en `derived-column` y en `having-clause`. Si aparece en una `derived-column` de un `select-stmt` sin `group-by-clause`, se agrupan todas las filas resultado en un mismo grupo.

Si las filas resultado están agrupadas, las `derived-column` y, de haberla, la `having-clause`, solo pueden referenciar columnas que no aparecen en la `group-by-clause` (o columnas en general si no hay `group-by-clause`) dentro de funciones de agregados.

```
having-clause = "HAVING" bool-expr
```

Indica que, de los grupos de filas, solo aparezcan en el resultado los que cumplan la condición.

6.4.4. Cláusula `ORDER BY`

```
order-by-clause = "ORDER" "BY" sort-spec *("," sort-spec)
sort-spec = [sid "."] id ["ASC" / "DESC"]
```

Una `order-by-clause` indica el orden de las columnas. Se usa la primera `sort-spec`, en caso de empate la segunda, etc., ordenando las filas según la columna de la tabla resultado indicada en orden ascendente (`ASC`, por defecto) o descendente (`DESC`). La mayoría de SGBDs interpretan la `order-by-clause` como un parámetro de `select-stmt` y no de `dql-stmt`.

6.4.5. Consideraciones de eficiencia

Se han de evitar anidamiento correlacionado y el ordenamiento por su coste. Para operar sobre relaciones, de mayor a menor elegancia[2] tenemos `IN`, `EXISTS` y `JOIN`, y de mayor a menor eficiencia[3] tenemos `IN` sin correlación, `JOIN` y `EXISTS`.

6.5. Álgebra relacional

El **álgebra relacional** es un lenguaje definido por Ted Codd en 1972 para expresar operaciones sobre relaciones.

Podemos ver una relación como una tupla (R, T, N) donde $T =: (T_1, \dots, T_n)$ es una tupla de conjuntos o **dominios**, $N =: (N_1, \dots, N_n)$ es una tupla de **nombres** distintos dos a dos y $R \subseteq (T_1 \dot\cup \{\texttt{NULL}\}) \times \cdots \times (T_n \dot\cup \{\texttt{NULL}\})$ es el **estado**. Decimos que las relaciones son **homogéneas** porque todos los elementos del estado son del mismo tipo.

Llamamos **grado** de una relación (R, T, N) a $\mathrm{gr}R \coloneqq |T|$ y **dominio** del atributo N_i a $\mathrm{dom}R_i \coloneqq T_i$. Las relaciones R y S son **compatibles en tipo** si $\mathrm{gr}R = \mathrm{gr}S$ y para $i \in \{1, \dots, n\}$, $\mathrm{dom}R_i = \mathrm{dom}S_i$.

Las relaciones se expresan por un nombre o como resultado de una operación. Si (R, T, N) y (S, U, M) son relaciones con grados respectivos n y m y

$$(a_1, \dots, a_n) * (b_1, \dots, b_m) \coloneqq (a_1, \dots, a_n, b_1, \dots, b_m),$$

los operadores son:

1. **Unión**: Si $T = U$, $R \cup S \coloneqq (R \cup S, T, N)$.
2. **Intersección**: Si $T = U$, $R \cap S \coloneqq (R \cap S, T, N)$.
3. **Diferencia**: Si $T = U$, $R - S \coloneqq (R \setminus S, T, N)$.
4. **Producto cartesiano ampliado**: Si N y M son disjuntos,

   $$R \times S \coloneqq (\{r * s\}_{r \in R, s \in S}, T * U, N * M).$$

   Cuando N y M no son disjuntos, sean $L : N \to N \amalg M$ y $R : M \to N \amalg M$ inclusiones, entonces $R \times S \coloneqq (R, T, L(N)) \times (S, U, R(M))$. Una forma de hacer esto es prefijar los nombres de atributos con el nombre de su tabla.
5. **Restricción**: Una **condición** sobre R se define por la gramática $S \to (S) \mid S \text{ AND } S \mid S \text{ OR } S \mid \text{NOT } S \mid TOC \mid TOT$, $O \to = \mid < \mid \leq \mid > \mid \geq \mid <>$, $T \to N_1 \mid \cdots \mid N_n$. Entonces, si C es una condición, $\sigma_C(R) \coloneqq (\{r \in R \mid C(r)\}, T, N)$, donde $C(r)$ se define con su significado habitual, teniendo en cuenta que $\texttt{NULL} \neq \texttt{NULL}$.

   La restricción es conmutativa: $\sigma_C(\sigma_D(R)) = \sigma_D(\sigma_C(R)) = \sigma_{C \text{ AND } D}(R)$.

[2] Según el criterio subjetivo de los profesores de la asignatura.

[3] Esto es muy dependiente de las optimizaciones del gestor de bases de datos.

6. **Proyección**: Sea $\iota : \{N_i\}_i \to \{1, \ldots, n\}$ la inversa de N, si M es una tupla de m nombres distintos dos a dos con $\{M_i\}_i \subseteq \{N_i\}_i$, entonces

$$\pi_M(R) := (\{(r_{\iota(M_1)}, \ldots, r_{\iota(M_n)})\}_{r \in R}, (T_{\iota(M_1)}, \ldots, T_{\iota(M_n)}), M).$$

7. **Reunión**: Una **condición de reunión** sobre R y S se define por la gramática $S \to T \mid T$ AND S, $T \to AOB$, $O \to = \mid < \mid \leq \mid > \mid \geq \mid <>$, $A \to N_1 \mid \cdots \mid N_n$, $B \to M_1 \mid \cdots \mid M_m$, salvo que si contiene $N_i \theta M_j$ con $O \to \theta$, debe ser $\mathrm{dom} R_i = \mathrm{dom} S_j$. Entonces, si C es una condición de reunión, $R \bowtie_C S := \sigma_C(R \times S)$. Si todos los operadores de C son la igualdad, hablamos de una **equi-reunión**. Definimos también $R \bowtie S := R \times S$.

   El producto cartesiano ampliado y la reunión son asociativas, y son conmutativas salvo orden de los atributos.

8. **Reunión natural**: Sea $\{j_1, \ldots, j_p\} := \{j \mid M_j \notin \{N_i\}\}$, si para $i, j \in \{1, \ldots, n\} \times \{1, \ldots, m\}$ con $N_i = M_j$ es $T_i = U_j$, entonces

$$R \divideontimes S := (\{r * (s_{j_1}, \ldots, s_{j_p}) \mid r \in R, s \in S, \forall i, j, (N_i = M_j \implies r_i = s_j)\}, T * U, N * M).$$

9. **Reunión externa**: Sea $N_k := \{\texttt{NULL}\}^k$. Definimos la **reunión externa izquierda** de R y S como $R] \bowtie_C S := R \bowtie_C S \cup (\{r \in R \mid \nexists s \in S \mid C(r, s)\} \times N_m)$, la **reunión externa derecha** como $R \bowtie [_C S := R \bowtie_C S \cup (N_n \times \{s \in S \mid \nexists r \in R \mid C(r, s)\})$ y la **reunión externa completa** como $R] \bowtie [_C S := (R] \bowtie_C S) \cup (R \bowtie [_C S)$.

10. **División**: Si $N := (N_1, \ldots, N_n, M_1, \ldots, M_m)$, entonces

$$R \div S := (\{r \mid \forall s \in S, r * s \in R\}, (T_1, \ldots, T_n), (N_1, \ldots, N_n)).$$

11. **Funciones de agregados**: Son SUMA, PRODUCTO, MÁXIMO, MÍNIMO y CUENTA, correspondientes a $\sum$, $\prod$, máx, mín y $|\cdot|$. Si O es el nombre de una de estas funciones, definimos la función de agregados $O_{N_i}(R) := O_{r \in R, r_i \neq \texttt{NULL}} r_i$.

    Dada una tupla $(F_1, \ldots, F_m)$ de funciones de agregados, ${}_{N_{i_1}, \ldots, N_{i_n}}\mathbf{F}_{F_1, \ldots, F_m}(R)$ es una relación que contiene, para cada clase de equivalencia $[r]$ de R respecto de la relación $r \equiv s : \iff (r_{i_1}, \ldots, r_{i_n}) = (s_{i_1}, \ldots, s_{i_s})$, una fila $(r_{i_1}, \ldots, r_{i_n}, F_1([r]), \ldots, F_m([r]))$. El tipo es $(T_{i_1}, \ldots, T_{i_n}, X_1, \ldots, X_m)$, donde X_i es el rango de F_i, y los nombres de los atributos son $(N_{i_1}, \ldots, N_{i_n}, F_1, \ldots, F_n)$.

Además, si la expresión *expr* toma el valor (S, T, M) con $\mathrm{gr} S = n$, la expresión $R(N_1, \ldots, N_n) \leftarrow$ *expr* establece R a $(S, T, \{N_1, \ldots, N_n\})$.

6.6. Cálculo relacional

El **cálculo relacional** es un lenguaje formal basado en la lógica predicados de primer orden que describe la información que se desea obtener de forma declarativa. Existen el **cálculo relacional de tuplas** (CRT), el que veremos, creado por Codd en 1972, y el **de dominios**, creado por Lacroix y Pirotte en 1977.

Las expresiones tienen forma

$$\{t_1.a_{11}, \ldots, t_1.a_{1m_1}, \ldots, t_n.a_{n1}, \ldots, t_n.a_{nm_n} \mid \mathrm{COND}(t_1, \ldots, t_n)\},$$

donde cada t_i es una **variable de tupla** y recorre una cierta relación y cada a_{ij} es un nombre de atributo de dicha relación.

Un **átomo** es una expresión de la forma $R(t_i)$, $t_i.a\,op\,t_j.b$, $t_i.a\,op\,c$ o $c\,op\,t_j.b$, donde R es una relación, t_i y t_j son variables de tupla, a es un atributo de la relación recorrida por t_i, b uno de la recorrida por t_j y c es una constante.

Una **fórmula bien formada** (f.b.f) es un átomo o, si α y β son f.b.f. y x es una variable, (α and β), (α or β), not(α), ($\alpha \to \beta$), $(\forall x)\alpha$ y $(\exists x)\alpha$. Una ocurrencia de una variable x es **ligada** si está en una fórmula de la forma $(\forall x)\alpha$ o $(\exists x)\alpha$, y es **libre** en caso contrario.

El valor de verdad de una fórmula puede ser TRUE (verdadero) o FALSE (falso). El de $R(t_i)$ es TRUE si y sólo si $t_i \in R$. Una fórmula bien formada es **cerrada** si toda variable es ligada, en cuyo caso su valor de verdad puede ser TRUE o FALSE, y es **abierta** en caso contrario, en cuyo caso representa una consulta.

$\mathrm{COND}(t_1, \ldots, t_n)$ es una f.b.f. con variables libres $t_1, \ldots, t_n$, y se cumple para las tuplas que, al ligar a las variables $t_1, \ldots, t_n$, la fórmula evalúa a TRUE.

Llamamos **dominio** de una expresión a la unión de los dominios de las relaciones que aparecen nombradas en la condición y las constantes que aparecen en la condición. Una expresión es **segura** si toda tupla para la que la condición se cumple tiene sus elementos en el dominio de la expresión. Entonces la expresión segura $\{T \mid \mathrm{COND}(t_1, \ldots, t_n)\}$ con dominio D se refiere al conjunto $\{T \mid t_1, \ldots, t_n \in \bigcup_{n\in\mathbb{N}} D^n \wedge \mathrm{COND}(t_1, \ldots, t_n)\}$. El cálculo relacional con expresiones seguras tiene la misma potencia que el álgebra relacional.

6.7. Índices

Buscar las filas de una tabla que satisfacen una condición es la operación más común en consultas SQL. La búsqueda secuencial implica un gran número de lecturas de **bloques** o **páginas**, las unidades a nivel físico en las que el SGBD almacena las filas de tablas.

Un **índice** es una estructura de datos auxiliar para acelerar el acceso a las filas de una tabla según el valor de un **campo de indexación**, generalmente una o más columnas. Contiene una serie de **entradas** o **registros** con cada valor de los existentes en el campo de indexación y un puntero al bloque del fichero de datos que contiene la fila con dicho valor. Las entradas están ordenadas según el valor de este campo[4], permitiendo hacer búsqueda binaria.

Se recomienda crear un índice cuando un campo se usa frecuentemente en condiciones de selección (`where-clause`) o de reunión, así como en todas las claves primarias y alternativas.

Los índices no forman parte del estándar SQL, pero existe un estándar de facto.

```
ddl-stmt =/ index-def / drop-index-stmt
index-def = "CREATE" "INDEX" sid "ON" sid "(" index-part *("," 
    index-part ")"
index-part = (id / expr) ["ASC" / "DESC"]
```

Un `index-def` crea un índice en una tabla, con ordenación similar a `ORDER BY`.[5] Además, se crea un índice por cada restricción `PRIMARY KEY` o `UNIQUE` con las columnas de la clave.

[4] Normalmente se usan variantes de árboles B.

[5] Algunas bases de datos permiten también índices parciales, en los que se añade una `where-clause` al final de la `index-def` y solo se indexan las filas que cumplen la condición.

```
drop-index-stmt = "DROP" "INDEX" sid
```

Elimina un índice. Esto puede hacerse porque ya no se espera realizar consultas basadas en el campo de indexación, o porque este no acelera las consultas al ser la tabla muy pequeña o contener muchas filas pero pocas entradas en el índice.

6.8. SQL del software privativo Oracle Database[6]

Cada comando del LDD realiza un `COMMIT` implícito antes y después de su ejecución, y también se hace un `COMMIT` implícito al salir correctamente de una herramienta como el software privativo Oracle SQL Developer o el software privativo Oracle SQL*Plus. Cuando la terminación es anormal o con error, se hace un `ROLLBACK` implícito.

Los índices se implementan con el estándar de facto.

6.8.1. Esquemas

El software privativo Oracle Database solo soporta un esquema por cuenta de usuario, que solo se puede borrar borrando la cuenta (no soporta `drop-schema-stmt`).

```
schema-def = "CREATE" "SCHEMA" "AUTHORIZATION" id
```

No hace nada. Además, en vez de `INFORMATION_SCHEMA` hay un *Data Dictionary*.

6.8.2. Tipos

```
type =/ raw-type / large-type / rowid-type
number-type =/ "NUMBER" ["(" uint ["," uint] ")"] /
    "BINARY_FLOAT" / "BINARY_DOUBLE"
string-type =/ "CHAR" ["(" uint ["BYTE" / "CHAR"] ")"] /
    "VARCHAR2" ["(" uint ["BYTE" / "CHAR" ] ")"] / "NVARCHAR2"
    "(" uint ")" / "LONG"
time-type =/ "TIMESTAMP" ["(" uint ")"] "WITH" "LOCAL" "TIME"
    "ZONE"
raw-type = "RAW" "(" uint ")" / "LONG" "RAW"
large-type = "BLOB" / "CLOB" / "NCLOB" / "BFILE"
rowid-type = "ROWID" / "UROWID" ["(" uint ")"]
```

`NUMBER` equivale a `NUMERIC`, con parámetros por defecto 38 y 0. `INTEGER` equivale a `NUMBER(38,0)`, `BINARY_FLOAT` a `REAL` y `BINARY_DOUBLE` a `DOUBLE PRECISION`. `CHAR(`*n* `CHAR)` equivale a `CHAR(`*n*`)` y `CHAR(`*n* `BYTE)` a `BIT(`*8n*`)`. `VARCHAR2` equivale a `VARCHAR` o a `BIT VARYING`, con las mismas consideraciones.

Los `raw-type` y `BLOB` contienen secuencias de bits, siendo `BLOB` preferible a `LONG RAW`. Los `large-type` contienen objetos grandes (*Large OBjects*), y `BFILE` referencia un objeto almacenado fuera de la base de datos.

[6] Si bien este es el SGBD que vemos en clase, lo cierto es que, según lo visto en clase, es bastante limitado (por ejemplo, en el tratamiento de claves ajenas y en conformidad con el estándar), y es difícil de instalar. Para un SGBD fácil de instalar y usar, véase SQLite. Para uno más acorde al estándar pero con muchas características adicionales, véase PostgreSQL.

6.8.3. Tablas

Se puede crear una tabla inicializada a los resultados de una consulta:

```
table-def =/ "CREATE" "TABLE" sid "AS" query
```

Hay más posibilidades en `ALTER TABLE`:

```
ddl-stmt =/ rename-stmt
rename-stmt = "RENAME" id "TO" id
alter-table-clause = "ADD" "(" column-def *("," column-def) ")"
    / "RENAME" "COLUMN" id "TO" id / "RENAME" "TO" id / "DROP"
    "COLUMN" id / "DROP" "(" id *("," id) ")" ["CASCADE"
    "CONSTRAINTS"] / "MODIFY" id (type / column-constraint)
    *column-constraint / alter-constraint-clause
```

`ALTER TABLE` *name* `RENAME TO` *newName* equivale a `RENAME` *name* `TO` *newName*, y cambia el nombre de una tabla. `ADD` tiene otra sintaxis y permite añadir varias columnas a la vez. `DROP COLUMN` tiene una sintaxis alternativa para eliminar más de una columna a la vez, y `CASCADE CONSTRAINTS` es como `CASCADE` en SQL. `MODIFY` cambia la definición de una columna.

```
alter-constraint = "ADD" 1*table-constraint / "MODIFY"
    ("CONSTRAINT" id / "PRIMARY" "KEY" / "UNIQUE" column-list)
    constraint-state ["CASCADE"] / "RENAME" "CONSTRAINT" id "TO"
    id / 1*drop-constraint-clause / enable-or-disable-clause
constraint-state = 1*(["NOT"] "DEFERRABLE" / "INITIALLY"
    ("IMMEDIATE" / "DEFERRED") / "RELY" / "NORELY" / "ENABLE" /
    "DISABLE" / "VALIDATE" / "NOVALIDATE")
drop-constraint-clause = "DROP" (("PRIMARY" "KEY" / "UNIQUE"
    column-list) ["CASCADE"] [("KEEP" / "DROP") "INDEX"] /
    "CONSTRAINT" id ["CASCADE"])
enable-or-disable-clause = ("ENABLE" / "DISABLE") ["VALIDATE" /
    "NOVALIDATE"] ("UNIQUE" column-list / "PRIMARY" "KEY" /
    "CONSTRAINT" id) ["CASCADE"] [("KEEP" / "DROP") "INDEX"]
```

`ENABLE` o `DISABLE` permiten activar o desactivar una restricción. Una clave primaria o alternativa referenciada por claves ajenas no se puede desactivar salvo que se indique `CASCADE`, en cuyo caso se desactivan también las claves ajenas.

```
drop-table-stmt = "DROP" "TABLE" [id "."] id ["CASCADE"
    "CONSTRAINTS"] ["PURGE"]
```

`CASCADE CONSTRAINTS` es como `CASCADE`, y `PURGE` libera el espacio ocupado por la tabla inmediatamente, sin situar la tabla y sus objetos dependientes en la papelera de reciclaje.

6.8.4. Reglas de integridad

La expresión en una restricción `CHECK` debe ser booleana y no puede:

- Contener subconsultas.

- Llamar a `SYSDATE`, `SYSTIMESTAMP`, `CURRENT_DATE`, `CURRENT_TIMESTAMP`, `DBTIMEZONE`, `LOCALTIMESTAMP`, `SESSIONTIMEZONE`, `UID`, `USER`, `USERENV` o funciones definidas por el usuario.
- Referenciar columnas de otras tablas.
- Contener constantes de tipo fecha no totalmente especificadas.
- Contener las pseudocolumnas `LEVEL`, `ROWNUM`, `NEXTVAL` y `CURRVAL`.

La cláusula `ON UPDATE` para claves ajenas no está soportada, con lo que siempre se asume `NO ACTION`, y `ON DELETE` solo puede ser `CASCADE` o `SET NULL`, y si no aparece se asume `NO ACTION`.

6.8.5. Vistas

```
view-def = "CREATE" ["OR" "REPLACE"] [["NO"] "FORCE"] "VIEW" sid
    [column-list] "AS" query ["WITH" "CHECK" "OPTION"
    ["CONSTRAINT" sid] / "READ" "ONLY"]
```

`OR REPLACE` permite cambiar la definición de la vista. `FORCE` crea la vista aunque no existan las tablas base o el propietario de la vista no tenga permisos sobre ellas. `WITH READ ONLY` hace que la vista no sea actualizable. `CONSTRAINT` *`sid`* da un nombre como restricción a la condición `WITH CHECK OPTION`.

```
drop-view-stmt = "DROP" "VIEW" sid ["CASCADE" "CONSTRAINTS"]
```

`CASCADE CONSTRAINTS` es como `CASCADE`.

6.8.6. Consultas

El operador `EXCEPT` se llama `MINUS`. `SELECT UNIQUE` es sinónimo de `SELECT DISTINCT`. Aunque esta característica es estándar, en el software privativo Oracle Database, la inclusión de una subconsulta en una `from-clause` se llama Oracle Online View.

Capítulo 7

Sistemas de bases de datos

7.1. Entorno

El entorno de la base de datos está formado por:

- Hardware.
- Software: SGBD, aplicaciones, sistema operativo y software de red.
- Datos, y metadatos en el catálogo del sistema.
- **Procedimientos**: Instrucciones y reglas para el diseño y uso de la base de datos y las aplicaciones.
- Personas o **actores**.

En empresas tenemos los siguientes actores:

1. **Administrador de la base de datos** (**ABD**), responsable de administrar los recursos del SBD: base de datos, SGBD y programas de acceso. Adquiere el software y hardware necesario; implementa la estructura de la base de datos y las restricciones de integridad; crea y modifica las estructuras de almacenamiento y métodos de acceso; concede o deniega permisos de acceso; define planes de copias de seguridad de los datos; garantiza el funcionamiento correcto del sistema, y proporciona servicio técnico.

2. **Diseñadores de bases de datos**: Interactúan con los futuros usuarios del sistema; recogen y comprenden sus requisitos; hacen el diseño conceptual, y eligen estructuras para representar y almacenar los datos. Para ello crean vistas que satisfacen los requisitos de cada grupo de usuarios (subconjuntos de la información en la base de datos) y crean un diseño final resultado de integrar las vistas.

3. **Desarrolladores de software**.

   a) **Analistas de sistemas**: Determinan las necesidades de procesamiento de los usuarios y especifican conjuntos de operaciones que las satisfacen.

   b) **Desarrolladores de aplicaciones**: Implementan las especificaciones en programas, que prueban, depuran, documentan y mantienen.

4. **Usuarios finales** o **clientes**.

   a) **Inexpertos**: Acceden de forma frecuente y repetitiva mediante aplicaciones que facilitan sus operaciones, y no son necesariamente conscientes de la existencia del SGBD.

   b) **Avanzados** o **sofisticados**: Familiarizados con la estructura de la base de datos y las funcionalidades del SGBD. Acceden de forma esporádica y distinta cada vez usando por ejemplo SQL.

Recuperar los datos de forma eficiente implica usar estructuras de datos complejas para representar la información, que se ocultan mediante niveles de abstracción. La **arquitectura en 3 niveles ANSI/SPARC** consta de:

1. **Nivel interno**: Descrito en un **esquema interno** con un modelo de datos físico. Define los tipos de registros, su secuencia física y las estructuras de almacenamiento y acceso. Es muy cercano al nivel físico, pero no trata con registros físicos como bloques o páginas ni con cilindros o pistas.

2. **Nivel conceptual** o **lógico**: Descrito en un **esquema conceptual** o **lógico** con un modelo conceptual o lógico. Define la estructura de toda la base de datos.

3. **Nivel externo** o **de vistas**: Descrito en uno o varios **esquemas externos** o **vistas** con un modelo conceptual o lógico. Cada uno define la porción de la base de datos que interesa a un grupo de usuarios. Varias vistas pueden solaparse entre sí.

Los SGBD no distinguen del todo los 3 niveles, pues suelen incluir detalles físicos en el esquema y usar el mismo modelo de datos para especificar las vistas y el esquema conceptual.

El ABD especifica la correspondencia entre los esquemas en el diccionario de datos del catálogo del sistema, dando a la base de datos una naturaleza autodescriptiva que permite al SGBD acceder a datos de cualquier aplicación.

La **independencia de datos** es la capacidad de modificar el esquema de un nivel sin tener que cambiar el esquema del nivel superior. Distinguimos:

1. **Independencia lógica de datos** en la **correspondencia externo/conceptual**: capacidad de modificar el esquema lógico sin alterar esquemas externos y programas, difícil.

2. **Independencia física de datos** en la **correspondencia conceptual/interno**: Capacidad de modificar el esquema interno sin alterar el esquema lógico y los programas, más fácil.

Con el modelo ANSI/SPARC, la modificación del esquema de un nivel provoca cambios en la correspondencia, no en el nivel superior, pero este aumenta el gasto en la compilación y ejecución de programas, reduce la eficiencia y supone actualizar constantemente las correspondencias.

Con el modelo actual, la mayoría de cambios en la estructura u organización de datos no implica cambios en los programas, al contrario que con el procesamiento de ficheros.

7.2. Funciones

Una SGBD proporciona un catálogo accesible para el usuario, servicios para la independencia de datos y servicios de integridad, entre otros.

También oculta los detalles de implementación física y proporciona un procesamiento de consultas eficiente mediante búferes, cachés de datos, técnicas de búsqueda y optimización y estructuras auxiliares como índices, con estructura de árbol o tabla *hash*. La elección de qué índices crear y mantener es responsabilidad del ABD.

Otra función es garantizar que las transacciones cumplen las propiedades **ACID**:

1. ***Atomicity*** (**atomicidad**): Se hacen todas las operaciones o ninguna.
2. ***Consistency*** (**consistencia**): Llevan la base de datos de un estado consistente a otro.
3. ***Isolation*** (**aislamiento**): Su ejecución es independiente del resto de transacciones, con lo que los cambios no son visibles para otras transacciones hasta que finaliza. Puede que no se imponga estrictamente, sino que haya niveles de aislamiento.
4. ***Durability*** (**durabilidad**): Si finaliza con éxito y es confirmada, sus cambios perduran aunque el sistema falle después.

Los **sistemas de procesamiento de transacciones** son sistemas con grandes bases de datos y muchos usuarios ejecutando transacciones. Requieren alta disponibilidad y respuesta rápida, por lo que varias transacciones pueden ejecutarse concurrentemente y acceder y actualizar los mismos datos.

Cada transacción tiene un **área de trabajo privada**, un área en almacenamiento primario (memoria principal) y secundario (disco magnético, cinta magnética, disco óptico, etc.) donde guarda los datos que lee o escribe.

Un **búfer de base de datos** es un conjunto de bloques de caché que contienen temporalmente los bloques de la base de datos requeridos por las transacciones. El SGBD carga los bloques que necesita leer en el búfer, escribe al búfer y decide cuándo volcar qué bloques modificados del búfer a disco.

Los SGBDs también suelen tener **servicios de autorización** y **seguridad** mediante **control de acceso selectivo**. Se da acceso solo a usuarios autorizados con cuentas protegidas con contraseña, y solo a ciertas partes de la base de datos para ciertas operaciones mediante restricciones de seguridad para cada cuenta, ejecutadas por el SGBD.

En general, los SGBDs también se integran con software de comunicaciones para que los usuarios puedan acceder a la base de datos de forma remota, y tienen utilidades adicionales para ayudar al ABD a administrar la base de datos:

1. De importación y exportación de datos, para cargar datos a partir de ficheros poco estructurados, descargarlos en ficheros de varios tipos e intercambiar información entre distintos SGBDs.
2. De monitorización o supervisión del uso, operación y rendimiento del sistema.
3. De análisis estadístico, para examinar las estadísticas de uso y rendimiento.
4. De reorganización de ficheros o índices.

5. De copia de seguridad.
6. De ordenamiento, compactación o compresión de ficheros.

7.3. Servicios de recuperación de fallos

Tipos de fallos en una SGBD:

1. **Fallo local previsto** por la aplicación, cancelación programada.
2. **Fallo local no previsto**: Error de programación.
3. **Fallo por concurrencia**: La **serializabilidad** de las transacciones es la equivalencia entre el orden de procesamiento de estas y algún orden secuencial. El servicio de control de concurrencia puede abortar una transacción porque incumple la serializabilidad o para romper un interbloqueo, y esta debe ser reiniciada más tarde.
4. **Fallo del sistema** o **caída suave**: Mal funcionamiento del hardware, software o la red que no daña el disco.
5. **Fallo del disco** o **caída dura**: Aterrizaje del cabezal de disco, soporte no legible, etc. Partes del disco pueden perder sus datos.
6. **Fallos físicos** y **catástrofe**s: Desastre natural como inundación, terremoto, incendio o apagón; robo, sabotaje, destrucción o corrupción de datos, de hardware, de software o de las instalaciones, intencionado o negligente.

Los 3 primeros tipos de fallos son **locales**, pues solo afectan a una transacción, y los otros 3 son **globales**, pues afectan a todas las transacciones en ejecución.

La **recuperación** tras un fallo que deja la base de datos en un estado inconsistente o sospechoso de serlo consiste en restaurar un estado previo al fallo, consistente y cercano al fallo.

Con un **fichero de bitácora**, **diario**, ***log***, ***journal*** o **registro histórico** podemos seguir la pista de la ejecución de cada transacción, y usar una **técnica de recuperación** para restablecer la base de datos a un estado consistente.

Al fichero de bitácora solo le afectan los fallos de tipo 5 y 6, y está formado por registros de los siguientes tipos:

- <INICIAR, *T*>: Inicio de la transacción *T*.
- <LEER, *T*, *x*>: *T* ha leído el valor del elemento *x*.
- <ESCRIBIR, *T*, *x*, *anterior*, *nuevo*>: *T* ha cambiado el valor del elemento *x* del *anterior* al *nuevo*.
- <COMMIT, *T*>: *T* ha finalizado con éxito y sus cambios están listos para confirmarse en disco.
- <ROLLBACK, *T*>: *T* ha sido anulada y sus cambios están listos para ser revertidos.

- `<REGISTRO DE COMPROBACIÓN, ...>`: Indica qué datos han sido ya escritos en cierto instante.

Cuando una transacción termina de ejecutar `COMMIT` o `ROLLBACK`, esto se debe haber anotado en la bitácora, los bloqueos deben haberse liberado y los cursores haberse cerrado. Una transacción llega a su **punto de confirmación** cuando termina de ejecutar `COMMIT` con éxito, y se dice entonces que la transacción está **confirmada**. En la recuperación, las operaciones de transacciones no confirmadas se deshacen en orden inverso de aparición, y a continuación, las operaciones de transacciones confirmadas se rehacen en orden de aparición.

Insertar una entrada en la bitácora supone leer su último bloque, actualizarlo y escribirlo de nuevo al disco, y para evitarlo se mantiene un **búfer de bitácora** con este bloque en que se escriben las entradas hasta que se llena. Se usa **escritura anticipada en bitácora** o **bitácora adelantada**, en la que no se escriben a disco los cambios hasta que se escriban a disco los correspondientes registros de bitácora, pues si se escribieran, podría caerse el sistema antes de registrar en bitácora los cambios hechos a la base de datos y por tanto estos no se podrían deshacer. Además, un `COMMIT` no se completa hasta que todas las entradas de bitácora de la transacción se hayan escrito a disco.

Con esto, habría que leer la bitácora completa en la recuperación. Para evitarlo, cada varios segundos o transacciones completadas, el SGBD marca un **punto de comprobación** o ***checkpoint***, en el que:

1. Suspende la ejecución de las transacciones.
2. Fuerza la escritura en disco del búfer de bitácora y de todo bloque modificado del búfer de base de datos.
3. Añade a la bitácora un `<REGISTRO DE COMPROBACIÓN>`, que indica, para cada transacción activa, su identificador y sus entradas primera y última.
4. Escribe en un **fichero especial de arranque** la dirección del registro de comprobación en la bitácora.
5. Reanuda la ejecución de las transacciones.

Con esto, la recuperación se hace con el algoritmo 1.[1]

7.4. Estructura general

Además del **gestor de base de datos** (**GBD**), el SGBD tiene:

- Un procesador de consultas, que recibe consultas de usuarios y las envía al GBD.
- Un preprocesador de comandos LMD, que se comunica con el procesador de consultas para compilar los comandos y que el programa los envíe al GBD.
- Un compilador de comando LDD, que compila las órdenes del ABD y las envía al gestor del catálogo, que modifica el catálogo a través del gestor de ficheros en coordinación con el GBD.

[1] Además, se puede ahorrar mucho espacio eliminando las entradas de la bitácora anteriores a la primera de las transacciones activas.

si *hay registro de comprobación* $(T_i, S_i, E_i)_{i=1}^n$*, donde* T_i *es un ID de transacción,* S_i *es su primera entrada y* E_i *es su última* **entonces**
  $s \leftarrow$dirección del registro;
  $a \leftarrow \{T_i\}_i$;
sinó
  $s \leftarrow 0$;
  $a \leftarrow \emptyset$;
fin
$C \leftarrow \emptyset$;
para *e entrada en la bitácora desde s* **hacer**
  si *e* =<INICIAR, T> **entonces** añadir T a A;
  sinó, si *e* =<COMMIT, T> **entonces** mover T de A a C;
fin
para *e entrada desde la última hasta* $\min(\{S_i\}_i \cap A)$ *o* 0 **hacer**
  si *e* =<ESCRIBIR, T, ...> *con* $T \in A$ **entonces** deshacer e;
fin
para *e entrada desde s hasta la última* **hacer**
  si *e* =<ESCRIBIR, T, ...> *con* $T \in C$ **entonces** rehacer e;
fin

Algoritmo 1: Algoritmo de recuperación por bitácora.

Cuando el GBD recibe un comando, comprueba la autorización y lo pasa al procesador de comandos, que lo pasa por el comprobador de integridad y el optimizador de consultas y envía una versión de este de bajo nivel al gestor de transacciones, que la pasa al planificador y este al gestor de datos. El gestor de datos consta del gestor de recuperación seguido del gestor del búfer, que se comunica con el gestor de ficheros del sistema operativo para manipular la base de datos a través de los métodos de acceso y búferes del sistema.

La base de datos consta de ficheros de datos, estructuras de acceso como índices y un catálogo el sistema, que contiene metadatos sobre estructura y organización, restricciones de integridad, autorización, estadísticas sobre los datos para optimización de consultas, etc.