Codd's Definition of Database Management Systems

A DBMS should provide the following functions:

- Data storage, retrieval and update
- User accessible catalog or data dictionary describing the metadata
- Support for transactions and concurrency
- Facilities for recovering the database should it become damaged
- Support for authorization of access and update of data
- Access support from remote locations
- Enforcing constraints to ensure data in the database abides by certain rules

Properties of Database Management Systems

- Physical Data independence Users of a database can ignore how their data is stored in practice.
- Logical independence users can be given a partial view of the data.
- Fourth Generation Language The database should be controlled (queries and update) through an interface where the users express their intention regardless of how the database actually computes it.
- Query optimization The queries and update should be automatically optimized to be as efficient as possible.
- Logical integrity The DBMS should verify that update keep data in a consistent state with regard to the constraints on the structure of data.
- Physical integrity The DBMS should try to stay coherent in the case of events like losing power, etc.
- Data sharing Multiple users can access the data while preserving the logical and physical integrity.
- standardized A DBMS should use a standardized interface so one could swap one DBMS vendor by another vendor without any major change to the code

Properties of RDBMS(Relational Database Management System)

- High decoupling between: 1. data model AND how it stored
2. queries AND how they are executed
- Allows for complex queries
- High optimization of queries with indexes
- Software that are reliable, stable, featureful
- Supports integrity constraints
- Can cover large or very large datasets (Gigabyte or even Terabytes)
- Supports transactions with ACID properties

Some standards for databases

- Relational simple but powerful model (tables)
- XML, recursive data, complex queries, used to be hyped.
- Json Documents, similar to XML but the fad moved to this one.
- Graph data is modeled as a graph, complex queries, very fashionable.
- Object complex hierarchical data model, inspired by OOP
- Key-Value simple queries and simple data model, performance oriented.
- Olap-cube oriented for performance of analytical queries

第一个 DBMS-1960s; standard Query Language(SQL)-1979

ACID properties

- Atomicity: A transaction block is either fully executed or completely canceled.
- Consistency (or Correctness): The resulting database is valid w.r.t to the integrity constraints.
- Isolation: The effect of two concurrent transactions is the same as if one was scheduled before the other.
- Durability: Once confirmed, a transaction cannot be rolled back.

CAP theorem

No cluster of computers can guarantee simultaneously 不能同时满足

- Coherence, i.e every successful read sees the latest write
- Availability, i.e. all queries are successful
- Partition Tolerance i.e. the system continues even if network messages are lost

Immediate consequence: When network failure, has to either cancel the current operations or proceed with the risk of being incoherent

weaknesses of traditional RDBMS

- Hard to scale to very very large dataset (the peta-byte)
- Hard to scale to several thousand queries per second
- Does not model inherently recursive data (eg. trees)
- ACID incurs noticeable overhead (latency, disk, CPU) due to locks and journalization
- Generally disk-bound for typical job

A Schema is composed of:

- Several tables or relations.
- Each relation has several columns or attributes.
- Each column has a type (INTEGER, BIGINT, VARCHAR, ...)

The data is stored as records or tuples into this table

Attributes are the describing characteristics or properties that define all items pertaining to a certain category applied to all cells of a column.

Tuple: The rows, instead, are called tuples, and represent data sets applied to a single entity to uniquely identify each item. Attributes are, therefore, the characteristics of every individual tuple that help describe its unique properties.

Relation schema: A set of attributes is called a relation schema. A relation schema is also known as table schema (or table scheme). A relation schema is the basic information describing a table or relation. It is the logical definition of a table. Relation schema defines what the name of the table is. This includes a set of column names, the data types associated with each column.

Relational schema may also refer to as database schema. It is the collection of relation schemas for a whole database. Relational or Database schema is a collection of meta-data. Database schema describes the structure and constraints of data representing in a particular domain. A Relational Schema can be described a blueprint of a database that outlines the way data is organized into tables. This blueprint will not contain any type of data. In a relational schema, each tuple is divided into fields called Domains.

Different types of integrity constraints

- key: When the sets of attributes A is a key constraint, it means we cannot have two tuples t1 and t2 such that t[A] = t2[A].
- Foreign-key given two relations R1 and R2, A → B is a foreign-key constraint between R1 and R2 when it means that for each tuple t1 in R1, then there exists a unique tuple t2 in R2 such that t1[A] = t2[B]. Note that there might exists t1' with t1'[A] = t2[B] and there might be a t2' [B] with no corresponding t1. It refers to the PRIMARY KEY in another table.
- PRIMARY KEY: is a specific choice of a minimal set of attributes (columns) that uniquely specify a tuple (row) in a relation (table).[a][1] Informally, a primary key is "which attributes identify a record," and in simple cases constitute a single attribute: unique NON NULL ID. More formally, a primary key is a choice of candidate key (a minimal superkey); any other candidate key is an alternate key. 只能有一个主键，但主键可以有多个列
- Superkey is a set of attributes that uniquely identifies each tuple of a relation. Because superkey values are unique, tuples with the same superkey value must also have the same non_key attribute values. that is, non-key attributes are functionally dependent on the superkey.
- Check Given a relation R, a check constraint is a boolean function f such that for each t ∈ R we have f (t) = True.

Relational Language

- R, for R a relation in S • RENAME(t,a,b) • DROP(t,a)
- FILTER(t,cond) • PRODUCT(t,t0) • UNION(t,t0) • DIFFERENCE(t,t0)

Overall Structure

- A database cluster contains users/groups and databases
- A database contains several schemas (the default one is public)
- A schema contains tables
- The same table name can occur in multiple schemas
- Can be qualified with the schema name
- Notion of search path to disambiguate unqualified names
- A table has a structure (also called its schema) and data (rows)

Naming tables and attributes

- Table names should be singular
- No accents, no special characters, underscores rather than spaces
- you can use double quotes for this, but discouraged
- Table and attribute names are not case-sensitive
- Except if using double quotes - still discouraged
- Probably have a column named id for the primary key (later)
- Several tables can have the same attribute name, but they will need to be disambiguated (e.g, R.id vs S.id)
- Avoid any reserved names (eg, end)
- Most important: consistency!

Basic PostgreSQL types

- BOOLEAN for Boolean values
- INT for integers (4-byte)
- SERIAL for an auto-incrementing identifier (4-byte), or AUTO INCREMENT with MySQL
- REAL for floating-point numbers (4-byte)
- NUMERIC for high-precision numbers (1000 digits)

- TEXT or VARCHAR: text
- VARCHAR(42): text of length at most 42
- BYTEA or BLOB for binary strings
- TIMESTAMP for date and time (can be WITH TIME ZONE), DATE, etc.
- Other: money, enumerated types (enums), geometric types, JSON and XML, network addresses, UUIDs, arrays..

Schema design with Entity-Relationship Diagrams

To decide which tables you should create for an application, you should

- Be very clear about what the goal of the application is!
    - Do not overlook this step!
    - often, schema design asks many tricky questions which data to manipulate, which assumptions are made, what should be possible or not..
    - On large projects, the database schema is often the central reference on which

data the application manages

- Formalize the logical schema, describing abstractly which data is managed
- Possibly, think about the operations that will be supported on this data (e.g., business processes)
- Implement the logical schema as a physical schema, i.e., concrete table definitions in a database
- check the resulting schema for problems (normalization)

Entity-relationship model(ER model)

- Entity-Relationship diagrams are a general model to present the logical schema of your application
- This is not pure science? necessarity a bit handwavy, and many variants/notations
    - Relates to object-oriented programming
    - Specificaly, to the Unified Modeling Language (UML)
- Basic notions
    - Entities (and entity-types), describing the "objects"
    - Relationships (and relationship-types), describing the "relationships" between

them

What are the goals of a good Schema design?

- Being complete, i.e., can represent everything that is needed
- Being clear to developers and as simple as possible
- Being precise: clear how to map actual business needs to data
- Not being too broad, i.e., correctly reflect constraints that are assumed
- Avoiding redundancy: make sure every data item is in one place
- Ensuring good performance (often linked to simplicity)

Basic Entity relationship Notions

Entity

Entity is a concrete object that we will have to manage. Examples:

- A person, a company, e.g., a customer, a supplier
- An actual object
- A location, a house, a building, a room..
- A file, a dataset, a data item
- An event
- An order, a request..

- Entities have attributes, e.g, name, size, date of birth, color, geographic coordinates, path, date, etc. The attributes of an entity-type can be sometimes subdivided, e.g, "address" becomes something like: number • street • extra info • building • floor • apartment number • city • post code. These are called composite attributes
- An entity-type is a type of entity, e.g, a "class" in software engineering
    - Customer, Supplier, Location, File, order, etc.
- All entities in the same entity-type have the same attributes

Attribute types

When we have an attribute we must think about its type: • String (which language? which text encoding?)• Integer• Decimal• Date/Time• Geographical coordinates, etc.

We must also think about:

- The domain of the attribute (which values are allowed?) • whether the attribute is mandatory (can we have no value?) • Which attribute(s) are the key that uniquely identifies the entity
    - There cannot be two different entities with the same values on all attributes
    - Either add the missing attributes, or add a surrogate key attribute

Two special kinds of attributes:

- **Derived attributes** can be deduced from other attributes

e.g, an "age" attribute can be deduced from a "date of birth" attribute

→ We will often not store the derived attribute, but compute it on the fly

Multi-valued attributes: there can be more than one value

e.g., email address, phone number..

→ We will often store these attributes in a separate table

Relationship

- A relationship connects two or more concrete entities

→ e.g "Customer 42 placed order 45"

→ e.g, "Professor Patricia supervised student John on topic 44"

- A relationship-type is a set of relationships with the same attributes and connecting the same entity-types

→ e.g, placesOrder, advises

- The possible participating entities are called roles

→ customer (Customer), order (order)

→ advisor (Professor), advisee (Student), topic (Topic)

→ Can have the same entity-type twice, e.g., "ismentoring" with mentor (Employee) and mentee (Employee)

- A relationship (and relationship-type) can also have attributes

→ e.g., date

画图：

Entities (formally entity-types) are often drawn in a rectangular box

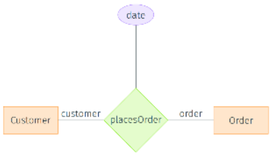
Attributes of the entity-type can be oval nodes, or lines in the box

Relationships (formaly relationship-types) are often drawn in a diamond box

Relationships are connected to the entities that are involved in them

Attributes are connected to the relationship

Roles are written on the edges connecting the relationship and entity



Cardinality constraints

For a given entity-type in a relationship-type, there can be cardinality constraints to describe if an entity can be:

- In no relationship
- In one relationship
- In multiple relationships

Beware of confusion:

- A given relationship always has one entity of each role!
- This is about the number of relationships to which a given entity participates
- Cardinality constraints apply per relationship (type), not across al relationships

Partial/total

Indicate whether 0 is acceptable or not:

- Total participation: 0 is not acceptable, every entity must be in a relationship

→ Represented by a double line in an ER diagram

- Partial participation (default): 0 is acceptable, some entities are not in a relationship

One/many to one/many

The key of the weak entity will be the key of the other entity in the identifying relationship plus a set of attributes called discriminator which is dash-underlined

Specialization and generalization

- A special kind of relationship: is-A
- Every professor is an employee
- Every employee is a person
- Lab sessions and lectures are classes
- We could represent, e.g., each professor with two entities (e.g., a professor entity and an employee entity), and have an is-A relationship between the two
- Sometimes more legible to write them with a "isA triangle"
- The subclass inherits attributes from the superclass
- Related to inheritance in object-oriented Programming
- Specialization top-down design process subdividing entities in subclasses
- Generalization: bottom-up design process regrouping entities sharing common attributes

Translating a ER model into schema

Translating entity:

Create one table per entity-type with all of the attributes

PS: 对于可能一个 attribute 有多个 value 的情况（如一个人可能有很多个电话）

Add an extra table with a foreign key for multi-valued attributes; Can also handle extra information

Translation of relationships

Create one table per relationship(用 foreign keys 链接 entities)

This is the proper solution, e.g., for many-to-many relationships

在做这些表的时候，key 的选择：

- In the general case of many-to-many relationships, the pair of identifiers
- For one-to-one one-to-many, many-to-one relationships, an identifier on the "many" side is enough
- Note: always possible to create a surrogate key

对于 One-to-many or many-to-one relationship 的情况，store the other objects and relationship attributes in attributes of the "many" side. 如老师指导学生，把老师写在学生 table 的指导教师这一个 attribute 下。若关系不是 total 的，则这个 attribute 可以为 null

对于 total one-to-one relationship, 可以全部写成一个表。

Normalization

First normal form if the data of every cell is an atomic type.

A functional dependency on a relation R is an assertion of the form A1 . . . An → B1 . . . Bm, where the Ai and Bj are attributes of R

Semantics: for any two tuples in R, if they agree on all of A1 . . . An then they agree on all of B1 . . . Bm

An violation of an FD A1 . . . An → B1 . . . Bm is two tuples that • Agree on (all) the attributes A1 . . . An • Disagree on (some of) the attributes B1 . . . Bm

2NF：数据表里的所有非主属性都要和该数据表的主键有完全依赖关系；如果有哪些非主属性只和主键的一部份有关的话，它就不符合第二范式

3NF：表中的所有数据元素不但要能唯一地被主关键字所标识，而且它们之间还必须相互独立，不存在其他的函数关系

Boyce-Codd Normal Form：如果对于关系模式 R 中存在的任意一个非平凡函数依赖 x->A，都满足 x 是 R 的一个超键，那么关系模式 R 就属于 BCNF。任何非主属性不能对主键子集依赖 BCNF disallows, for instance: • FDs between non-key attributes (attributes outside the key) • FDs from a strict subset of the key attributes

Closure of A1 . . . An: All attributes B such that A1 . . . An → B holds Call this B01 . . . B0p: it contains in particular B1 . . . Bm

View: You can define a view to represent the result of a complex query:

CREATE VIEW movie__with_actor AS SELECT DISTINCT Movie.id, title FROM Movie, Actor__in_movie WHERE Movie.id = Actor_in__movie.movie;

View 的好处

- Logical independence: you can change the definition of the view in an application without changing the rest of the code
- can be used to restrict access rights (only allow users to see a specific view)
- can be switched easily to a materialized view for performance

How to make the view refresh automatically? Workaround: • make the materialized view a regular table • Define triggers to update it in the right way whenever the underlying tables are changed

Stored procedures：是在大型数据库系统中，一组为了完成特定功能的 SQL 语句集，它存储在数据库中，一次编译后永久有效，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。

- For triggers (see later)
- To factor some application logic in the database for consistency across applications
- For performance (execute code closer to the data, can be written in C)

Trigger 触发器是 SQL server 提供给程序员和数据分析员来保证数据完整性的一种方法，它是与表事件相关的特殊的存储过程，它的执行不是由程序调用，也不是手工启动，而是由事件来触发，比如当对一个表进行操作（insert, delete, update）时就会激活它执行。

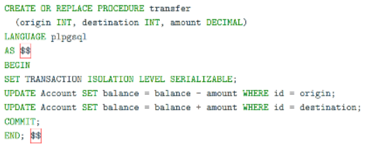
Possible uses:

- Complex consistency check, or normalization/reformatting
- Recomputing auxiliary tables, automatically creating dependent data
- manually updating an aggregate (e.g., a sum)
- manually log database operations

Table inheritance Tables can inherit from multiple tables • Deleting a parent table cascades to the tables that inherit from it • warning: uniqueness constraints and keys do not take inheritance into account! 如果不加 only，那么 select 之后会出现被继承的表里的数据

事务（Transaction）一般是指要做的或所做的事情。数据库事务（Database Transaction）是由 SQL 语句组成的逻辑处理单元。一个事务是一条 SQL 语句或一组 SQL 语句，不管是一条还是一组，都是可以一个事务。事务中的 SQL 语句就是一个整体，其中的 SQL 语句要么都执行，要么都不执行。MySQL 事务常用于操作量大，复杂性高的数据，避免因误操作而出现数据信息异常，确保了数据的一致性和完整性。

Default every query (SELECT, INSERT, etc.) is a transaction • We can manually define a transaction block with BEGIN .. COMMIT • Start a transaction with BEGIN, and issue queries • To perform the transaction, use COMMIT • To abort the transaction, use ROLLBACK • To define a savepoint, use SAVEPOINT label • To roll back to a savepoint, use ROLLBACK TO SAVEPOINT label



```
CREATE OR REPLACE PROCEDURE transfer
  (origin INT, destination INT, amount DECIMAL)
LANGUAGE plpgsql
AS $$
BEGIN
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE Account SET balance = balance - amount WHERE id = origin;
UPDATE Account SET balance = balance + amount WHERE id = destination;
COMMIT;
END; $$
```

Challenges with single transactions

To correctly support transactions (one at a time) we must:

- prepare the effects of the transaction and atomically commit them
- make sure the commits are durable, even if the hardware fails
- Be able to revert the effects of the transaction
- With save points, be able to revert its partial effects

Strongest ACID guarantees: serializability, 是最严格的隔离级别,所有事务按照次序依次执行，因此，脏读、不可重复读、幻读都不会出现。此时事务具有最高的安全性，但是由于事务是串行执行，所以效率会大大下降，应用程序的性能会急剧降低。

并发控制 (Concurrency Control)(基于锁/基于时间戳)

Also supports explicit locking in transactions (in addition to these mechanisms)

数据库中事务并发控制的正确性，也就是我们常说的 **isolation**. More restrictive isolation means worse performance, more failures, but less inconsistency problems

Replication 的思想是将数据在集群的多个节点同步、备份，以提高集群数据的可用性。• partition the data if it is large • do load balancing to use multiple servers • evaluate a query on multiple servers in parallel • have failover servers for high availability

Evaluation algorithms

- JOIN(a,JOIN(b,c)) → JOIN(JOIN(a,b),c) Associativity
- JOIN(a,b) → JOIN(b,a) commutativity
- FILTER(JOIN(a,b),f) → JOIN(FILTER(a,f),b) Distributivity when f operates on attributes in a

count different kinds of operations separately:

- number of seeks × tseek
- number of blocks read sequentially × tread
- number of blocks written × twrite
- number of CPU operations × tcpu
- size of block sblock (typically a few KB)

Index

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息

- Hash for equality tests only

[Pros] • Generally the fastest index (but with a small margin) • Especially good on large datasets with complex types like strings • Small space overhead

[Cons] • At least two seeks required • Cannot retrieve data in sorted order •

Algorithm	Average	Worst case
Space	O(n)[1]	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

- B-tree for arbitrary comparisons

B-tree of order m is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every non-leaf node (except root) has at least ⌈m/2⌉ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains k − 1 keys.
5. All leaves appear in the same level and carry no information.

[Pros] • Quite fast (especially on simple types) • Retrieve data in order with very few seeks • can be used for prefixes / subkeys; a B-tree on (x, y) can be used to test whether a given x exists or retrieve the corresponding y.

[Cons] • Larger number of seeks required • Larger space overhead (especially for complex data types)

B 树的删除：1）如果当前需要删除的 key 位于非叶子结点上，则用后继 key 覆盖要删除的 key，然后在后继 key 所在的子支中删除该后继 key。此时后继 key 一定位于叶子结点上，删除这个记录后执行第 2 步 2）该结点 key 个数大于等于 Math.ceil(m/2)-1，结束删除操作，否则执行第 3 步。3）如果兄弟结点 key 个数大于 Math.ceil(m/2)-1，则

父结点中的key下移到该结点，兄弟结点中的一个key上移，删除操作结束。否则，将父结点中的key下移与当前结点及它的兄弟结点中的key合并，形成一个新的结点。原父结点中的key的两个孩子指针就变成了一个孩子指针，指向这个新结点。然后当前结点的指针指向父结点，重复上第2步。有些结点它可能即有左兄弟，又有右兄弟，那么我们任意选择一个兄弟结点进行操作即可。

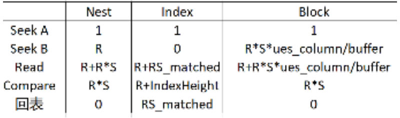
Algorithm	Average	worst case
Space	O(n)	O(n
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

- GIN(Generalized mverted Index)for case-insensitive search, n-grams, prefixes, JSON field lookup, etc.
- GiST(Generaized Search Tree) for non ordered data (e.g. geographical data, date intervals, etc.)
- Bitmap indexes : For each value v, store a bit array of length n (the number of records) where the i-th bit is set when the record has value v 位图索引适合只有几个固定值的列，如性别、婚姻状况、行政区等等，而身份证号这种类型不适合用位图索引。此外，位图索引适合静态数据，而不适合索引频繁更新的列

Inverted Index:

Implementation: Typically a B-tree or a hash index giving for each i the set of e such that i ∈ xe. Usage: They can be used for indexing JSON fields or getting records where some word appears in a string.

Joins can be computed with many different algorithms (外表大小R，内表S)



	Nest	Index	Block
Seek A	1	1	1
Seek B	R	0	R*S*ues_column/buffer
Read	R+R*S	R+RS_matched	R+R*S*ues_column/buffer
Compare	R*S	R+IndexHeight	R*S
回表	0	RS_matched	0

- Nested loop join

SELECT * FROM unicode u1, unicode u2

- Block nested loop joins
- Index-join (Index Nested Loop Join)
- Sort-merge join. Get both inputs in sorted order; Advance in both inputs by increasing order; Output matching couples

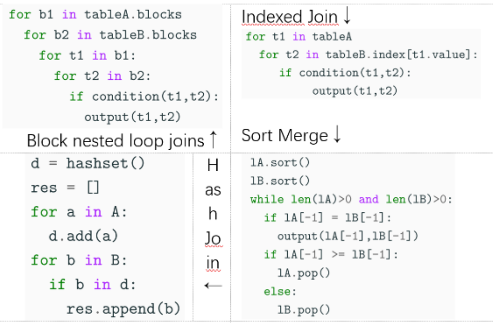
SELECT * FROM unicode u1, unicode u2 WHERE u1.numeric=u2.numeric ;

或 SELECT * FROM unicode u1, unicode u2 WHERE u1.comment=u2.comment

- Hash join. More generally, for a join with a condition C containing a set of equalities x1 = y1 ∧ . . . ∧ xk = yk :

SELECT * FROM unicode u1, unicode u2 WHERE u1.codePoint=u2.lowercase

- Indexed-join: Pre-compute and index the join result. Allows for very fast join but not without drawbacks



```
for b1 in tableA.blocks
  for b2 in tableB.blocks
    for t1 in b1:
      for t2 in b2:
        if condition(t1,t2):
          output(t1,t2)
```

Block nested loop joins ↑

Indexed Join ↓

```
for t1 in tableA
  for t2 in tableB.index[t1.value]:
    if condition(t1,t2):
       output(t1,t2)
```

Sort Merge ↓

```
lA.sort()
lB.sort()
while len(lA)>0 and len(lB)>0:
  if lA[-1] = lB[-1]:
    output(lA[-1],lB[-1])
  if lA[-1] >= lB[-1]:
    lA.pop()
  else:
    lB.pop()
```

```
d = hashset()
res = []
for a in A:
  d.add(a)
for b in B:
  if b in d:
    res.append(b)
```

Hash Join ←

The various postgres operators

Full scan: SELECT * FROM myTable

If you only select a handful of rows, PostgreSQL will decide on an index scan – if you select a majority of the rows, PostgreSQL will decide SeqScan. But when need to read too much for an index scan to be efficient but too little for a sequential scan? use a bitmap scan.

- SeqScan explore the full table SELECT * FROM unicode
- Index Scan explore the relevant part of a table using an index(极少的行)

SELECT * FROM unicode WHERE codepoint='0000'

- Index only Scan explore the index

SELECT * FROM unicode WHERE codepoint<'0000' ;

- Bitmap Heap Scan like a SeqScan but might skip some of the disk pages using a bitmap index SELECT * FROM unicode WHERE charname='something'
- Bitmap index Scan builds a bitmap with an index. It can read too much for an index scan to be efficient but too little for a sequential scan 同上
- Bitmap or / Bitmap And builds a bitmap as the or/And of two bitmaps

SELECT * FROM unicode WHERE numeric IS NOT NULL or codepoint = '0000';

ELECT * FROM unicode WHERE numeric IS NOT NULL AND charname < 'B'

- Filter, Nested loop, Hash join, Merge join, self explanatory.

SELECT * FROM unicode WHERE comment IS NOT NULL ;

- materialize we need to talk about the execution... Database engines prefer to "stream" tuples rather than computing everything.

练习题

SQL

- supplier(sid, sname, address), indicating supplier details,
- Parts(pid, pname, color), indicating the details of each part, and
- Catalog(sid, pid, cost), listing the prices charged for parts by suppliers.

1. Find the names of suppliers who supply some red part.

SELECT S.sname FROM Suppliers S, Parts P, Catalog C WHERE P.color='red' AND C.pid=P.pid AND C.sid=S.sid

2. Find the sids of suppliers who supply some red or green part.

SELECT C.sid FROM catalog C, Parts P WHERE (P.color = 'red' OR P.color = 'green') AND P.pid = C.pid

3. Find the sids of suppliers who supply some red part or are at 221 Packer Street.

SELECT S.sid FROM Suppliers S WHERE S.address = '221 packer street' OR S.sid IN (SELECT C.sid FROM Parts P, catalog C WHERE P.color='red' AND P.pid = C.pid)

4. Find the sids of suppliers who supply some red part and some green Part. SELECT C.sid FROM Parts P, catalog C WHERE P.color = 'red' AND P.pid = C.pid AND EXISTS (SELECT P2.pid FROM Parts P2, Catalog C2 WHERE P2.color = 'green' AND C2.sid = C.sid AND P2.pid = C2.pid)

5. Find the sids of suppliers who supply every Part.

SELECT C.sid FROM Catalog C WHERE NOT EXISTS (SELECT P.pid FROM Parts P WHERE NOT EXISTS (SELECT C1.sid FROM Catalog C1 WHERE C1.sid = C.sid AND C1.pid = P.pid))

6Find the sids of suppliers who supply every red or green part.

SELECT C.sid FROM Catalog C WHERE NOT EXISTS (SELECT P.pid FROM Parts P WHERE P.color = 'red' OR P.color = 'green' AND NOT EXISTS (SELECT C1.sid FROM Catalog C1 WHERE C1.sid = C.sid AND C1.pid = P.pid))

7. Find the sids of suppliers who supply every red part or supply every green part. SELECT C.sid FROM catalog C WHERE (NOT EXISTS (SELECT P.pid FROM Parts P WHERE P.color = 'red' AND (NOT EXISTS (SELECT C1.sid FROM catalog C1 WHERE C1.sid = C.sid AND C1.pid = P.pid)))) OR (NOT EXISTS (SELECT P1.pid FROM parts P1 WHERE P1.color = 'green' AND (NOT EXISTS (SELECT C2.sid FROM catalog C2 WHERE C2.sid = C.sid AND C2.pid = P1.pid))))

8. Find pairs of sids such that the supplier with the first sid charges more for some part than the supplier with the second sid.

SELECT C1.sid, C2.sid FROM Catalog C1, Catalog C2 WHERE C1.pid = C2.pid AND C1.sid <> C2.sid AND C1.cost > C2.cost

9. Find the pids of parts supplied by at least two different suppliers.

SELECT C.pid FROM catalog C WHERE EXISTS (SELECT C1.sid FROM Catalog C1 WHERE C1.pid = C.pid AND C1.sid <> C.sid)

10. Find the pids of the most expensive parts supplied by suppliers named Yosemite Sham.

SELECT C.pid FROM Catalog C, Suppliers S WHERE S.sname = 'Yosemite Sham' AND C.sid = S.sid AND C.cost >= ALL (SELECT C2.cost FROM Catalog C2, Suppliers S2 WHERE S2.sname = 'Yosemite Sham' AND C2.sid = S2.sid)

Relational Language

- Room (Name, Time, MovieTitle) • Movie (MovieTitle, Director, Actor)
- Producer (ProducerName, MovieTitle) • Seen (Spectator, MovieTitle)
- Like (Spectator, MovieTitle)

Q5: Among the actors who produced at least one movie? S5: actors = DROP(DROP(movie, MovieTitle), Director); producers = DROP (Producers, MovieTitle); DROP(FILTER(PRODUCT(actors, producers), ProducerName=Actor), ProducerName)

Q10: who are the spectators watching all the movies? S10: movieNames = DROP(Movie, [Director, Actor]); spectators = DROP (Seen, MovieTitle); allPairs_ms = PRODUCT(movieNames, spectators); missingPairs = DIFFERENCE(allPairs_ms, Seen); DIFFERENCE(spectators, missingpairs)

TP0 4: 国家信息数据库

name varchar(50): continent varchar(60): area decimal(10,0): population decimal(11,0): capital varchar(60):

- Return the country name and population of France, Germany, and Italy (in that order): SELECT name, population FROM world WHERE name IN ('France', 'Germany', 'Italy')
- Select the name, population, and area of all countries that are either large in population or large in size, but not both. Specifically, select countries that have at least 100 million inhabitants OR have an area of at least 3 million square kilometers, but not both. Order the results by name :

-SELECT name, population, area FROM world WHERE population >= 100000000 XOR area >= 3000000 ORDER By name

- Select the country names and capital names where both names have the same length exclude the cases where country name is same as the capital

-SELECT name, capital FROM world WHERE LENGTH(name)=LENGTH(capital) AND NOT name = capital ORDER BY name

- Return the name and population of the 5 countries with the greatest population.

SELECT name, population FROM world ORDER BY population DESC LIMIT 5 (若从第k个开始, OFFSET k)

- Return the name of Asian countries with a new column dense containing "yes" for countries with density at least 100 inhabitants per square kilometer, and "no" otherwise. Sort the results to have first the dense countries, then the non-dense countries, and then order the countries in each group by name

-SELECT name,'Yes' AS 'dense' FROM world WHERE continent = 'Asia' AND popuation/area >= 100 UNION SELECT name,'no' AS 'dense' FROM world WHERE continent = 'Asia' AND population/area < 100

- Return the continent names and, for each continent, the number of countries, total population and total area of the countries in that continent.

-SELECT continent, COUNT(name), SUM(population),SUM(area) FRom world GROUP BY continent ORDER BY continent

- Compute, for every continent name, its population density, and the average of the population densities of its countries.两个值不一样

SELECT * FROM (SELECT continent, SUM(population)/SUM(area) as density1 FROM world GROUP By continent) as group1 LEFT JOIN (SELECT continent ,group2.density2 as density3 FROM (SELECT continent, SUM(population/area)/COUNT(*) AS density2 FRom world GROUP BY continent) as group2) as group3 ON group1.continent = group3.continent

- Return a table with a column alpha containing the first letter of country names (ordered alphabetically) and a column total with the total popuation of countries whose name starts with that letter.

SELECT SUBSTR(name, 1, 1) AS alpha, SUM(population) AS total FROM world GROUP By alpha

- compute, for every continent name, a count of how many countries in the continent are strictly more populous that the continent's largest country (by area). Order the results by continent, and do not omit results where the count is zero. SELECT continent, SUM(group2.density2) / COUNT(group2.name) FROM (SELECT continent, name, population/area AS density2 FROM world) as group2 Group by continent

TP01-2 movie, casting, and actor

Movie(id, title, yr,director) casting(movie_id, actor__id, ord) Actor(id,name)

- Compute the years where the actor "Rock Hudson" participated to strictly more than one movie, along with the number of movies to which he participated on that year, sorted by decreasing number of movies, then by ascending year.

-SELECT yr,COUNT(table3.id) FROM (SELECT id,name FROM actor WHERE name = 'Rock Hudson') AS table1 LEFT JOIN (SELECT movieid, actorid FROM casting) AS table2 ON table1.id = table2.actorid LEFT JOIN (SELECT yr,id FROM movie) as table3 ON table3.id = table2.movieid GROUP BY yr HAVING COUNT(table3.id)>1 ORDER By COUNT(table3.id) DESC, yr ASC

- Limit the movies released no later than 1930, and to the actor names starting with A, B, or C. two actors X and Y are challengers if x was the leading actor in a movie where Y appeared and Y was the leading actor in a movie where x appeared. compute all pairs X, Y of challengers (with X < Y, in alphabetical order of x and then of Y).

-WITH oldmov AS (SELECT id, yr FROM movie WHERE yr <=1930),
aactor AS (SELECT id, name FROM actor WHERE name LIKE 'A%' OR name LIKE 'B%' OR name LIKE 'C%')

-SELECT DISTINCT A1.name, A2.name FROM aactor AS A1, aactor AS A2, casting AS C1a, casting AS C1b, casting AS C2a, casting AS C2b, oldmov AS Ma, oldmov AS Mb WHERE A1.id = C1a.actorid AND A1.id = C1b.actorid AND A2.id = C2a.actorid AND A2.id = C2b.actorid AND C1a.movieid = C2a.movieid AND C1b.movieid = C2b.movieid AND C1a.ord = 1 AND C2b.ord = 1 AND C1a.movieid = Ma.id AND C1b.movieid = Mb.id AND A1.name < A2.name
ORDER BY A1.name, A2.name

- In which movie title did Alain Delon and Catherine Deneuve appear together?

-SELECT title FROM movie, casting AS C1, casting AS C2, actor AS A1, actor AS A2 WHERE A1.id = C1.actorid AND A2.id = C2.actorid AND C1.movieid = movieid AND C2.movieid = movieid AND A1.name = 'Alain Delon' AND A2.name = 'Catherine Deneuve'

- Find the only actor who appeared in all Star Wars movies.

_ SELECT name FROM actor WHERE NOT EXISTS (SELECT id FROM movie WHERE TITLE LIKE 'star wars%' AND NOT EXISTS (SELECT * FROM casting WHERE movieid=movieid AND actorid=actor.id))

- Find the only actor which only appeared in movies where Harrison Ford appeared, and appeared in strictly more than one movie.

-SELECT name FROM actor, casting, movie WHERE casting.actorid = actor.id AND casting.movieid = movie.id AND name <> 'Harrison Ford' AND NOT EXISTS (SELECT id FROM movie, casting AS C1 WHERE C1.movieid = movie.id AND C1.actorid = actor.id AND NOT EXISTS (SELECT 1 FROM casting AS C2, actor AS A2 WHERE A2.name = 'Harrison Ford' AND C2.actorid = A2.id AND C2.movieid = C1.movieid)) GROUP BY name HAVING COUNT(movie.id) > 1

TP02,建立一个游戏商品交易数据库

Type: id,name

Player: id,name,money

Possesion: Id,type(外键 type),owner(外键 player),possession

Q. Allow a player to mark one of their item as buyable with a given price,

UPDATE possession SET price = <price>

Q: Alow a player to buy cheapest item of a given type from the marketplace

START TRANSACTION

SELECT id,price,owner as curowner FROM possession

WHERE buyable IS NOT NULL ORDER BY price LIMIT 1 ;

UPDATE player SET money = money + price WHERE id = curOwner ;

UPDATE player SET money = money - price WHERE id = <buyer> ;

UPDATE possession SET owner = <buyer> WHERE id = idobject

COMMIT

SELECT column_name(s) FROM table_name WHERE condition AND|OR condition

ALTER TABLE table_name ADD column_name datatype REFERENCES Filming(id)

ALTER TABLE Movie ALTER COLUMN test TYPE int USING test::integer;

ALTER TABLE Movie RENAME COLUMN test To test2;

ALTER TABLE movie DROP COLUMN test; 删除列是在这里的！

ALTER TABLE Movie RENAME TO Movie2;

DROP COLUMN column_name

SELECT column_name AS column_alias FROM table_name; SELECT column_name FROM table_name AS table_alias

SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2(包括前后)

CREATE DATABASE database_name

CREATE TABLE table_name(column_name1 data_type, (id SERIAL PRIMARY KEY, title VARCHAR, tstart DATE CHECK (tstart > '1895-01-01'), tend DATE, CHECK (tstart < tend));

CREATE INDEX index_name ON table_name (column_name) (Duplicate values are allowed)

CREATE INDEX movie_tags ON movies USING gin (tags);

CREATE UNIQUE INDEX index_name ON table_name (column_name) (Duplicate values are not allowed)

CREATE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition

DELETE FROM table_name WHERE some_column=some_value 用来删除行

DELETE FROM table_name (Note: Deletes the entire table!!); DELETE * FROM table_name (Note: Deletes the entire table!!)

DROP DATABASE database_name 整个全部删除

DROP INDEX table_name.index_name (SQL Server); DROP INDEX index_name ON table_name (MS Access); DROP INDEX index_name (DB2/Oracle); ALTER TABLE table_name DROP INDEX index_name (MySQL)

DROP TABLE table_name 整个全部删除

IF EXISTS (SELECT * FROM table_name WHERE id = ?) BEGIN --do what needs to be done if exists END ELSE BEGIN --do what needs to be done if not END

SELECT column_name, aggregate_function(column_name) FROM table_name WHERE column_name operator value GROUP BY column_name HAVING aggregate_function(column_name) operator value(常与 COUNT(), MAX(), MIN(), SUM(), AVG() 一起用)

SELECT column_name(s) FROM table_name WHERE column_name IN (value1,value2,.)

INSERT INTO table_name VALUES (v1, v2, v3,..); INSERT INTO table_name (column1, column2, column3,..) VALUES (v1, v2, v3,.)

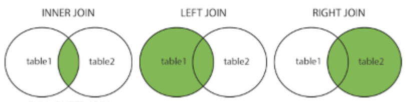
INSERT INTO table2 (column1, column2, column3, ..) SELECT column1, column2, column3, ... FROM table1 WHERE condition;

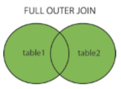
SELECT column_name(s) FROM table_name1 INNER JOIN table_name2 ON table_name1.column_name = table_name2.column_name

SELECT column_name(s) FROM table_name1 LEFT JOIN table_name2 ON ...

SELECT column_name(s) FROM table_name1 RIGHT JOIN table_name2 ON ...

SELECT column_name(s) FROM table_name1 FULL JOIN table_name2 ON ...





SELECT column_name(s) FROM table_name WHERE column_name LIKE pattern

% :zero, one, or multiple characters, (_) one single character. [abcde]=[a-e][! abc]

SELECT column_name(s) FROM table_name ORDER BY column_name [ASC|DESC], column_name2 [ASC|DESC] LIMIT 5

SELECT DISTINCT column_name(s) FROM table_name (return only distinct (different) values)

SELECT SUBSTR("SQL Tutorial", 5, 3) AS ExtractString; (Extract a substring from a string (start at position 5, extract 3 characters):)

SELECT MIN/AVG/MIN/COUNT(column_name) FROM table_name WHERE condition;

SELECT * INTO new_table_name [IN externaldatabase] FROM old_table_name (copies data from one table into a new table)

SELECT column_name(s) INTO new_table_name [IN externaldatabase] FROM old_table_name

SELECT TOP number|percent column_name(s) FROM table_name

TRUNCATE TABLE table_name 删除表里数据而不删除表本身

SELECT column_name(s) FROM table_name1 UNION SELECT column_name(s) FROM table_name2

SELECT column_name(s) FROM table_name1 UNION ALL SELECT column_name(s) FROM table_name2 (allow duplicate values)

UPDATE table_name SET column1=value, column2=value,.. WHERE some_column=some_value

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

WHERE NOT condition. 或者是 City NOT IN; NOT Like; NOT在列名的后面

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30
END AS QuantityText
FROM OrderDetails;
```

comments, with -- or /* .. */

\l to list databases \c database change database \dt list tables \d table show details about a table

PostgreSQL types : BOOLEAN , INT for integers (4-byte) SERIAL for an auto-incrementing identifier (4-byte), or AUTO INCREMENT with MYSQL REAL for floating-point numbers (4byte) NUMERIC for high-precision numbers (1000 digits) • TEXT or VARCHAR: text • VARCHAR(42): text of length at most 42 • BYTEA or BLOB for binary strings • TIMESTAMP for date and time (can be WITH TIME ZONE), DATE, etc

<>不等于 ^XOR %Module